記　号　処　理　60－4
プログラミング—言語・基礎・実践—　2－4
（1991. 6. 21）

Futures and Multiple Values in Parallel Common Lisp

田中 朋之　　渦原 茂

日本アイ・ビー・エム (株) 東京基礎研究所
〒102 千代田区三番町 5-19

多くの共有メモリ型の並列Lispでは並列性の記述にfuture式を用いている。本稿ではCommon Lispの多値機能にfutureを導入した場合の問題について考える。futureと多値機能を共存させるためには1つのfutureオブジェクトに複数の値を格納できるようにする必要がある。ところがそのままこの方法を用いると、プログラムにfutureを挿入した場合としなかった場合とで返す値の数が異なってしまう場合があり、Common Lispで定義される多値の意味を変えてしまう。この問題を解決するためにmv-context法とmv-pフラグ法の2つの方法を提案する。この2つの方法はマルチプロセッサ・ワークステーションTOP-1上の並列Lisp、TOP-1 Common Lispにおいて実現した。

Futures and Multiple Values in Parallel Common Lisp

TANAKA Tomoyuki　　UZUHARA Shigeru

IBM Research, Tokyo Research Laboratory
5-19, Sanbancho, Chiyoda-ku, Tokyo 102, JAPAN

The future construct is used in many shared-memory parallel Lisp systems to express concurrency. We consider the impact of introducing the future construct to the multiple value facility of Common Lisp. 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 that how such a 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. To overcome this problem, we present two methods which we call the mv-context method and the mv-p flag method. 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.

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 of Common Lisp. We briefly examined the problem and outlined our approach in [5]. 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 [5].

TOP-1 Common Lisp is a parallel modification of Kyoto Common Lisp (KCL) [8] featuring a real-time multiprocessor garbage collector [7]. For an overview of TOP-1 Common Lisp please see [5],

2 Futures and multiple values in Lisp

2.1 Futures

A *future* was first used in the Lisp language in Multilisp [2]. 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 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

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. It is 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 [3]. 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`. If there are more values produced than requested, the excess values are simply ignored. If there are less values produced than requested, the unpresent values defaults 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 [1]. We deal with the problem of futures and multiple values in Scheme in [6].

2.3 The goal: the coexistence of futures and multiple values

We have stated earlier that 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.

3 The implementation of futures with multiple values

3.1 The problem with the straightforward implementation

We observe that 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.

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)
```

`make-future` is an internal function which 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, and sets `resolved-p` to true. Exactly one value is stored regardless of the number of values 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)
  ;; FUTURE may or may not be a future.
  (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 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.

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*.

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)))
```

`mv-context` is an internal function that 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 calls the following `mv-touch` function when it's 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 which does not have exactly one future value, in other words, the first future which has either multiple number (2 or more, or 0) of values, or one non-future value.

```
(defun mv-touch (future)
  ;; 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))))))
```

This mechanism is implemented in a prototype version of TOP-1 Common Lisp

3.3.1 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.

This is sometimes possible even when the second `future` is not lexically within the first one. 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 do not need to be created.

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

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

```
        #:g001))
    ((multiple)
     (if (evaluating-for-future-p)
         (eval-set-future-m (future-evaluating-for) <form>)
         (let ((#:g001 (make-future)))
              (process-funcall
               #'(lambda () (eval-set-future-m #:g001 <form>)))
              #:g001))))
```

Two internal functions `evaluating-for-future-p` and `future-evaluating-for` are used in the expansion. `evaluating-for-future-p` returns if the form `(future <form>)` is being evaluated for a future, and `future-evaluating-for` returns the future for which evaluation is being done. At run-time these two piece of data must be available for evaluation of each form.

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 `touch`.

```
(defun mv-touch (future)
  ;; FUTURE is a future.
  ;; returns the "last" 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)
```

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. 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 the 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, fields in a future contains the `mv-p` flag instead of the `mv-context` field. The flag indicates that the future is capable of returning

multiple values. At run-time, 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)
  ;; 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))))))
```

4 Conclusions

We examined the problems involved in introducing the future construct to the multiple value facility of Common Lisp, and presented two methods of imple-

menting futures with multiple values: the `mv-context` method and the `mv-p` flag method. We have 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] Halstead, R.H. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS.* 7, 4 (Oct. 1985), pp. 501–538.

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

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

[5] 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.

[6] Tanaka T. and Uzuhara S. Futures and Multiple Values in Parallel Lisp. TRL Research Report, forthcoming.

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

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