Premature Return
- Another Interpretation of the Future Construct -
(An Extended Abstract)

Taiichi Yuasa
Department of Information and Computer Science
Toyohashi University of Technology
Toyohashi 440, Japan

Abstract

Several parallel Lisp languages such as MultiLisp[1], MultiScheme[2], and TOP-1 Common Lisp[3] use the *future* construct as the primitive parallel construct. Evaluation of a `future` expression requires spawning a subprocess. For relatively fine-grained parallel applications, the time for process creation affects the overall performance of the program. In this paper, we propose another interpretation of the future construct, called the *premature-return*, which is useful to reduce the overhead of process creation. With this interpretation, the caller process of a `future` expression keeps evaluating the expression while another process will be created to execute the program that follows the `future` expression. Although the premature-return is more efficient than the conventional interpretation in many cases, it is sometimes less efficient. In order to avoid such inefficient situations, we propose the use of an additional construct together with the premature-return.

1 The future construct

The future construct[1] has the following format.

(future E)

where E is an arbitrary form (a Lisp expression). When evaluated, this form immediately returns with an object called *future*, and creates a subprocess that will take care of the task of evaluating E (see Figure 1, where the down arrows indicate the control flow, and the right arrow indicates the process creation). When the value of E is obtained, that value replaces the future object and the execution of the subprocess terminates. In other words, the future is *instantiated* with the value. (The left arrow in Figure 1 indicates the instantiation.) On the other hand, the caller process of the `future` form keeps going without waiting for the value of E until it needs (or *touches*) the value of E. When the process needs the value of E, if the future is not instantiated yet, then the process will be suspended until the subprocess finishes evaluating E. (The dotted line in Figure 1 indicates the process suspension.) Thus the future construct allows concurrency between the "producer" of a value and the "consumer".

2 The premature-return interpretation

We now introduce our interpretation of the future construct called the *premature-return*. In this interpretation, the subform E in the form (`future` E) is evaluated by the process which calls

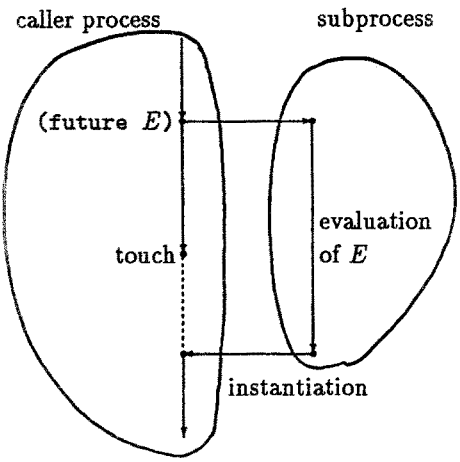

Figure 1: The future construct

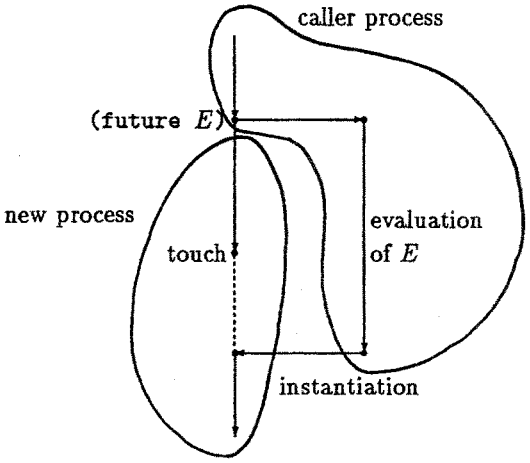

Figure 2: The premature-return

the future form, and a new process is created to execute the rest of the main task that follows the future form (see Figure 2).

Process creation is a heavy task. In particular, for fine-grained applications, the time for process creation affects the overall performance of the program. In Figure 1 and Figure 2, we ignored the effect of process creation. If we take the overhead of process creation into consideration, we obtain Figure 3(a) from Figure 1 and Figure 3(b) from Figure 2. Here, we assume that a process creation takes a constant time P, in order to simplify the discussion.

In the conventional interpretation of the future construct (Figure 3(a)), the evaluation of E starts at time P after the call of future. Supposing that the evaluation of E takes time t_2, the future object will be instantiated at time $P + t_2$ after the call of future. On the other hand, in the case of the premature-return (Figure 3(b)), the evaluation of E starts immediately after the call of future and thus the future object will be instantiated at time t_2. That is, the instantiation occurs earlier in the case of the premature-return.

Now, suppose in Figure 3(a) that the caller process touches the future object at time t_1. In the case of the premature-return, the execution following the call to future will start at time P, and the future object will be touched at time $P + t_1$. This means that, if $t_1 < t_2$, then the waiting process will resume earlier in the case of the premature-return. The gain is P in this situation. Note that this situation always happens when the main task is suspended for future instantiation.

On the other hand, if the main task is never suspended for future instantiation, i.e., if the subtask finishes its computation before the main task needs the future value, then the main task proceeds faster in the conventional interpretation than in the premature-return. In this case, the loss by the premature-return is P. Therefore, in these simple cases, we cannot say that the premature-return is more efficient than the conventional interpretation.

Let us consider more general cases where several processes are created to perform the computation. Figure 4 illustrates the situation where the main task creates a subtask and the subtask further creates another subtask, assuming that both the main task and the first subtask are suspended for future instantiation. This figure indicates that the gain of using the premature-return is $2P$. In general, if subtasks are nested n levels and if all tasks except the last are suspended for future instantiation, the gain of using the premature-return will be nP. On the other hand, if no tasks are suspended for future instantiation, the loss will be P independently of the nested levels.