# A New Transformation based on Process-Message Duality for Concurrent Logic Languages

**Kouichi Kumon**
Institute for New Generation Computer Technology
Mita-Kokusai Bldg 21F, Mita 1-4-28, Minato-ku, Tokyo 108, Japan
kumon@icot.or.jp

**Keiji Hirata**
NTT Basic Research Laboratories
3-1, Morinosato Wakamiya, Atsugi-shi, Kanagawa 243-01, Japan
hirata@nefertiti.ntt.jp

## Abstract

In a concurrent logic language, mapping from the objects in a programmer's mind to actual processes and messages often results in less-than-optimum performance. One reason for this is that an incorrect prediction of the dynamic behavior and the resulting irrelevant mapping make the execution threads short and thus result in frequent suspensions. Also, good performance may be less important than the ease of writing and reading programs. Although in typical language processing systems for a concurrent logic language, shorter execution threads are likely to cause inefficiency, only few attempts extend the thread length by improving the scheduling policy of processing systems.

We are proposing a new technique for extending thread length, by exchanging the roles of the processes and messages of a source program; we call this technique a *duality transformation*. This technique improves program performance without the need for modifying the language processing systems. Processes and messages in a concurrent logic language can perform the same roles, making it is possible to transform processes to messages, and vice versa. Furthermore, the notion of process-message duality not only improves efficiency, but also gives rise to new programming methodologies, which allow meta-level programming without meta-language facilities. In this paper, we discuss the concept of duality of processes and messages, details of the transformation to a dual program, and some actual applications of this transformation.

## 1 Introduction

When a programmer writes a program in a concurrent logic language, he/she is faced with making the important choice of which of the objects in his/her mind are to be mapped to processes and which should be mapped to messages. Thus, user programs often employ an object-oriented programming style, where a series of messages are sent to a perpetual process [8, 9]. We call this the message-oriented programming style.

However, a programmer's choice may result in poor performance, the reasons perhaps being that the mapping of objects to processes and messages is poorly suited to the execution environment, that is, the programmer's prediction of the dynamic behavior of his/her program is wrong, or achieving ease of writing and reading programs degrades efficiency. In general, since language processing systems for concurrent logic languages (the Monaco system[10], the PIM system[7] and the KLIC system[1] and others [12, 2, 5]) assume that, during execution many shorter threads, resulting from suspension, are the most likely cause of inefficiency. Therefore, the above systems have attempted to reduce the switching overhead incurred by suspension and resumption. On the other hand, only few attempts extend the thread length for message-oriented programs by improving the scheduling policy of processing systems[11].

We are proposing a new method where, in the rewriting of a source program, the roles of processes and messages are exchanged to improve the efficiency. As a result, despite no modifications being made at the language processing system level, our method can yield the same effect as improving the scheduling policy. Since processes and messages in a concurrent logic language work in the same way as a data container, it is possible to transform processes to messages, and vice versa. Hence, we can say that processes and messages are dual to each other. Furthermore, the notion of process-message duality provides us not only with efficiency improvement, but also enables new programming methodologies, that allow meta-programming without the need for meta-language facilities.

Section 2 of this paper discusses the notion of duality of processes and messages in the concurrent logic language, FGHC. Section 3 presents the procedure of transformation to a dual program. Section 4 shows some actual applications of the transformation to efficiency improvement and new programming methodologies. Section 5 concludes this paper.

## 2 Duality in a Concurrent Logic Language

### 2.1 Message-Oriented Programs and Process-Oriented Programs

When a programmer writes a program in a concurrent logic language, he/she has to use the notion of messages and processes. To discuss the meanings and roles of messages and processes, we first define a message-oriented program and a process-oriented program. These two types of programs are distinguished by the way in which they initiate their execution: message sending or predicate invocation.

**Message-Oriented Program:** A sequence of messages is sent to a perpetual process. Upon receiving it, the process is activated and updates its internal state, and may send newly created messages to other perpetual processes. After that, the process returns to the suspended state [8, 9]. Figure 1 is a schematic diagram illustrating the behavior of a simple message-oriented program. In this figure, the shaded ovals represent suspended processes,

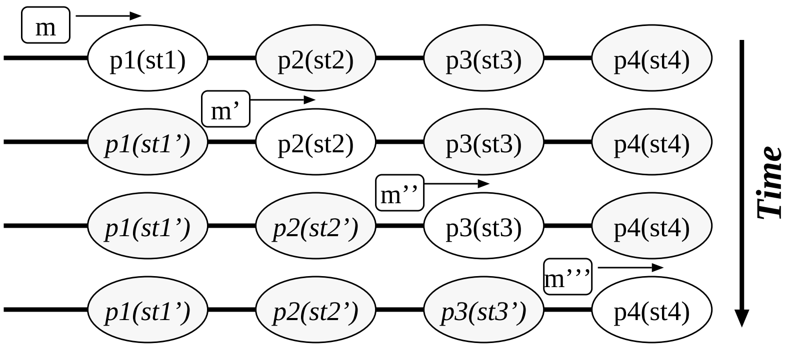


Figure 1: Behavior of Message-Oriented Program

while the white ovals represent active processes. Before message `m` arrives, all perpetual processes are suspended. Then, `m` activates process `p1`, which sends the new message `m'` to successive process `p2`. After that, `p1` is suspended again. As a result, many suspensions and resumptions occur during the execution of this type of program.

**Process-Oriented Program:** A predicate is invoked to process the bulk of the data. As the predicate is being executed, it may invoke other predicates to process data, and may create new data. If the previous example is rewritten in a process-oriented manner (Figure 2), predicates may be suspended less frequently, because enough data may already exist to execute process `m`. In this figure, the rounded rectangles represent the data lists to

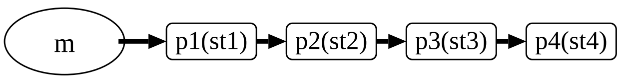


Figure 2: Behavior of Process-Oriented Program

be input to process `m`.

In message-oriented programs, we can think of messages as being active entities, with the processes waiting to be fed the messages. On the other hand, in process-oriented programs, the processes are active entities, that take in and process the data.

## 2.2 Duality of Messages and Processes

Before discussing the duality of processes and messages, we will first describe the execution of predicates by examining a sample program (Figure 3). We use FGHC, a concurrent logic language, in the following discussion.

At first, we look into how execution proceeds until predicate `p` commits. During the execution of the body of `top`, the message `[m|n]` is generated by active unification with variable `Mi` (`%a`), and predicate `p` is called (`%b`).

```
top:- true |
        Mi=[m|n],                          %a
        p(Mi,st,Mo).                       %b
p(Mi,ST,Mo):-                              %1
    Mi=[M|Mis],                            %2
    ST=st, M=m |                           %3
        Mn=m', STn=st',                    %4
        Mo=[Mn|Mos], r(Mis,STn,Mos).       %5
```

Figure 3: Sample Program for Illustrating Duality

This predicate invocation passes the argument values to the variables occurring at the head of the definition (`%1`). Then, the passive unification at `%2` decomposes a message packet, for which the message container is a list cell (`[ | ]`). Finally, it is checked whether the process state `ST` and the received message `M` for committing (`%3`). Here, we regard `%1` and `%2` as being the basic contribution to clause selection and `%3` as being an additional selection. The information required for the body execution of predicate `p` is:

- The data `ST`, which is carried as a process state of `p` (extracted at `%1` in Figure 3).
- The data `M`, which is carried by a message container `[ | ]` (extracted at `%2` in Figure 3).

At this point, we may exchange the carriers of the information (`ST` and `M`), thus obtaining the program shown in Figure 4. In this transformation, data structure `[ | ]` is read as `cons( , )`. The commented numbers in this

```
top:- true |
        cons(Mi,m,n),                      %a'
        Mi=p(st,Mo).                       %b'
cons(Mi,M,Mis):-                           %2'
    Mi=p(ST,Mo),                           %1'
    ST=st, M=m |                           %3
        Mn=m', STn=st',                    %4
        Mo=[Mn|Mos], r(Mis,STn,Mos).       %5
```

Figure 4: Transformed Program

program indicate the correspondence with the lines in the original program shown in Figure 3. In Figure 4, `cons` initiates the calculation and carries `m` as its process state (`%a'`), while `p` acts as a message container and brings `st` as the message contents (`%b'`). Correspondingly, the predicate callee and the message decomposer should be transformed (`%1'`, `%2'`). Note that it makes no difference whether the body execution of predicate `p` is initiated by a message or by a predicate call, but the body execution could possibly be transformed in a similar way (`%4`, `%5`).

We have already seen that the original program treats `cons` (*i.e.*`[ | ]`) as a message carrier and `m` as its contents. In the transformed program, they are treated as the process and process states, respectively. The original program in this example and the transformed program behave in much the same way, because the transformation changes the execution behavior of the program without changing of the meaning of the program. We define this relation as *dual*, and summarize this observation in the following table:

| | | |
|---|---|---|
| Process Invocation | *Caller* | *Callee* |
| | Predicate Call in Body | Head in Definition |
| | $\updownarrow$ *dual* | $\updownarrow$ *dual* |
| Message Sending | *Sender* | *Receiver* |
| | Active Unification | Passive Unification |

The next section discusses the details of these transformation rules.

# 3 Transformation to Dual Programs

## 3.1 Streams

A stream is the basic and useful data structure used by a concurrent logic language. We define a stream in an FGHC program as follows:

- A stream has a tree structure (no loops and no node sharing).
- Each node may have a distinct label (functor symbol).
- All nodes are single-referenced by their parent nodes. This means that there are no external references to the intermediate nodes of a tree.

We then have two kinds of streams; one where all nodes consist of messages and another where all nodes are processes. In a message stream (a stream consisting of messages), a predecessor directly references its successors. In a process stream, each reference is represented by a variable, shared by a predecessor and its successor.

The following two program examples show a message stream (`ms`) and a process stream (`ps`), respectively. Note that these two streams have an identical structure.

```
ms(S) :- true | S=a(1,R1), R1=b(R2,N), N=nil, R2=b(2,3).
ps(S) :- true | a(S,1,R1), b(R1,R2,N), nil(N), b(R2,2,3).
```

In the definition of `ms`, variables `R1`, `R2` and `N` refer to the successor nodes. On the other hand, in the definition of `ps`, each stream node, which is realized by a process, shares variables `R1`, `R2` and `N` with the successor nodes; there are always two references to the shared variables.

## 3.2 Restricted FGHC Program

To mechanically transform FGHC programs to dual ones, first, the programs must be translated into a *normal form*. Here, *normal form* means that the following restrictions are applied:

**(1)** There are no nested structures; that is, the entire data structure is expressed in a flat form by using '='. Consequently, at the top level of arguments of literals and function symbols, only variables and atoms occur.
**(2)** Passive unification for one level of a structure is performed in a clause.
**(3)** A predicate which is target of the transformation has, at most, one stream input.
**(4)** Variables, which are bound to the stream structure, are treated in a one-writer, one-reader manner.
**(5)** A reader process must read every created stream structure.

Here, *stream* refers not only the message stream but also the process stream.

Restrictions **(1)** $\sim$ **(2)** can be textually (statically) checked but, in general, the precise checking of restrictions **(3)** $\sim$ **(5)** is difficult to realize textually. The aliasing, modes, types, size and cost analyses enabled by abstract interpretation can be used for the checking [3]. We intend to pursue this in the future. Restriction **(5)** is necessary to avoid deadlock of the transformed program. An example concerning the duality between unread data and deadlock processes will be presented in Section 4.2. The following is an example of a restricted FGHC program.

```
p(S,T) :- S=cons(X,Ss), X > 1 | T=cons(X,Ts), p(Ss,Ts).
p(S,T) :- S=nil | T=nil.
```

Also, the program segments for the stream representation (`ms`, `ps`) in the previous section are written in restricted FGHC.

Many practical FGHC programs can be translated into restricted ones. Even if a predicate has more than one input stream, it is not necessary to transform all of these. Instead, you can select only the most appropriate stream for transformation. We shall use this method in Section 4.1.2. But, once an input stream has been selected, all the possible generators and consumers of the stream must be transformed. As a result, it may become necessary to transform more than one input stream simultaneously. In other cases, an input stream variable may be used for more than one level of passive unification.

In both cases, to realize **(2)** and **(3)**, fresh intermediate predicates can be introduced, after which an original predicate takes the form of a one stream-input and a one-level decomposition of the stream. Consider the following sample program:

```
add(X,Y,Z):- X=i(A,Xs),Y=i(B,Ys) |
    S := A+B, Z=i(S,Zs), add(Xs,Ys,Zs).
```

By introducing a new predicate `add2`, this definition is translated into a restricted FGHC program:

```
add(X,Y,Z) :- X=i(A,Xs) | add2(Y,A,Xs,Z).
add2(Y,A,Xs,Z) :- Y=i(B,Ys) |
    S := A+B, Z=i(S,Zs), add(Xs,Ys,Zs).
```

To achieve **(4)**, we may duplicate an active unification, provided the meaning does not change. Suppose that the following program is given:

```
h :- true | X=p, q(X), r(X).
```

Variable X has multiple references: one writer and two readers. Then, we can rewrite this without changing its meaning:

```
h :- true | X=p, Y=p, q(X), r(Y).
```

Moreover, we may introduce some syntactic support to encourage a one-writer one-reader manner [4, 5]. However, the duplication and syntactic support do not cover all the cases needed to satisfy restriction **(4)** (Section 4.2).

You will notice that there are some kinds of programs which cannot be translated into restricted FGHC. Such programs include non-deterministic predicates, such as merger. Therefore, dual merger does not exist in our current framework. We will discuss this facet in a forthcoming paper.

### 3.3 The Transformation Procedure

This section presents the procedure for transforming a restricted FGHC program to a dual one, using the sample programs shown in Figure 5. In the

```
top :-sg(17,S),sc(S,0). %a1       top :-sg(17,S),S=sc(0). %b1

sg(N,So) :- N > 0  |               sg(N,So) :- N  > 0 |
        So=c(N,Sos),    %a2                c(So,N,Sos),    %b2
        N1 := N-1,                         N1 := N-1,
        sg(N1,Sos).                        sg(N1,Sos).
sg(N,So) :- N =< 0 |               sg(N,So) :- N =< 0 |
        So=nil.         %a3                nil(So).        %b3

sc(Si,T) :-             %a4        c(Si,X,Sis) :-          %b4
        Si=c(X,Sis) |   %a5                Si=sc(T) |      %b5
        Tn:=T+X,                           Tn:=T+X,
        sc(Sis,Tn).     %a6                Sis=sc(Tn).     %b6
sc(Si,T) :-             %a7        nil(Si) :-              %b7
        Si=nil      |   %a8                Si=sc(T) |      %b8
        print(T).                          print(T).
```

(a) Message Stream (b) Process Stream

Figure 5: Transformation to Dual Program

figure, sg represents a stream generator, while sc represents a stream consumer. Program (a) in the left-hand column generates and consumes a message stream, while (b) does the same for a process stream. Both streams form a list of integer elements in descending order. In this example, the stream generators have their single internal states, N, and also the consumers have T as their internal states.

The first step of transformation is to determine the stream variable at the top level that connects a stream generator and a stream consumer. In this case, the variable is S at %a1 or %b1.

The second step is to determine the variable used for stream generation within a stream generator. Here, this is the second argument of the stream generator, So. Next, if So is bound to a message stream (%a2, %a3), these active unifications are transformed to process invocations (%b2, %b3). Otherwise, the inverse transformation (from process invocation to active unification) is performed. Here, it is sufficient to take only the body part into account.

The third step is to correspondingly transform the consumer program. In the guard part, the left- and right-hand sides of the neck operator (':-') are exchanged with respect to the variable for stream input (Si). Thus, the correspondence of the lines in Figure 5 is: %a4↔%b5, %a5↔%b4, %a7↔%b8, %a8↔%b7. At the same time, the form of message decomposition is rewritten into the form of the process callee (e.g. %a5→%b4) and vice versa. Finally, in the body part, the invocation of a stream consumer is transformed: %a1↔%b1, %a6↔%b6.

Note that, since a restricted FGHC program after transformation still belongs to restricted FGHC, the transformed program can also be transformed to a dual program. By applying the above transformation procedure, a *dual dual* program, which is identical to the original, is produced.

# 4 Applications

## 4.1 Performance Improvement Obtained By Applying Duality

This section highlights the efficiency improvement gained by the transformation to a dual program, taking two programs as examples: a binary tree search program and (the kernel part of) a life-game program. All timings are measured using one processing element of the PIM/p system[7, 6] and the KLIC system[1] (Version 1.010, running on a SparcStation 10/30). The PIM/p system is a parallel inference machine developed by ICOT. It employs tagged architecture and shared memories. User programs in KL1 are compiled into its RISC-like native instruction set. The KLIC system is a KL1 implementation for UNIX systems, which has already been ported onto several commercial UNIX systems. First, the KLIC system compiles user programs written in KL1 into C programs. Next, the system compiles the C code and links the object code with the KLIC runtime libraries.

We recognized from our experiments that certain types of programs run faster if they are first transformed. Currently, the transformation is done by hand. We think that it will be practical to formalize and automate the transformation procedure that is informally presented in Section 3.3. However, while we also consider it also feasible to determine all the stream variables (i.e. all the points that can be rewritten), it may prove difficult to determine to which stream variables the duality transformation should be applied to improve the program performance. At present, we believe that using a user's preferences will provide a practical solution to this problem.

### 4.1.1 Binary Tree Search

**Program Description:** A binary tree search program[11] represents tree nodes by using the 5-ary predicate which receives the search and update messages for searching for a specified key, and for updating the value for a specified key (Figure 6). We say that the tree is represented by *process tree*,

```
nt([],              _, _, L,R) :- true | L=[], R=[].        %1
nt([search(K,V)|Cs], K, V1,L,R) :- true |                   %2
           V=V1, nt(Cs, K, V1, L, R).
nt([search(K,V)|Cs],K1,V1,L,R) :- K<K1 |                    %3
           L=[search(K,V)|L1], nt(Cs,K1,V1,L1,R).
nt([search(K,V)|Cs],K1,V1,L,R) :- K>K1 |                    %4
           R=[search(K,V)|R1], nt(Cs,K1,V1,L,R1).
nt([update(K,V)|Cs],K,_,L,R) :- true   | nt(Cs,K,V,L,R).    %5
nt([update(K,V)|Cs],K1,V1,L,R) :- K<K1 |                    %6
           L=[update(K,V)|L1], nt(Cs,K1,V1,L1,R).
nt([update(K,V)|Cs],K1,V1,L,R) :- K>K1 |                    %7
           R=[update(K,V)|R1], nt(Cs,K1,V1,L,R1).

t([]             ) :- true | true.                          %8
t([search(_,V)|Cs]) :- true | V=undefined,t(Cs).            %9
t([update(K,V)|Cs]) :- true | nt(Cs,K,V,L,R),t(L),t(R).     %10
```

Figure 6: Ueda's Binary Tree Search Program (Original)
(process tree, list stream notation)

and that the `search` and `update` messages are carried by a *list stream* (the stream whose constructors are list cells, i.e. `[search(K,V)|Cs]`). In general, the stream carrier, 2-ary cons, is used only for pointing to the subsequent message carrier. If the message itself can point to the next message, such as `search(K,V,Cs)`, the stream carrier, 2-ary cons, can be omitted. We call this representation of a stream the *functor stream*. The functor stream is more efficient than the list stream, since the functor stream requires fewer unification operations. We measured the performance of dual programs in terms of both streams.

In this example, we prepare two types of message generators. When the first sends a message, it does not check whether the previous message has already been processed. Therefore, we call this the *no-wait* generator. The second defers the sending of a new message until all previously sent `search` messages have been processed. The second message generator is called the *acknowledge-wait* generator.

**Duality Transformation:** Prior to being transformed to its dual program, first, the source program is translated into restricted FGHC. Thus, all list cells are explicitly treated as the 2-ary `cons` functor, after which we obtain the 3-argument `cons` predicate. Figure 7 shows the transformed program. Each clause in Figure 6 is numbered by comments `%1`∼`%10`, the same-numbered comments in Figure 7 indicating the corresponding clauses. The transformed program uses the 4-ary functor symbol, `nt`, to represent a tree node (*functor tree*), the 2 arguments of `nt` being used to hold the key and the value. The remaining 2 arguments are used to point to the left and right branches of the tree node. This dual representation of the binary tree is also straightforward and easy to understand.

```
nil(C) :- C=nt(_ ,_ ,L,R) | nil(L),nil(R).                     %1
nil(C) :- C=t             | true.                              %8

cons(C,search(K,V),Cs) :- C=nt(K ,V1,L,R)       |              %2
          V=V1, Cs=nt(K,V1,L,R).
cons(C,search(K,V),Cs) :- C=nt(K1,V1,L,R), K<K1 |              %3
          cons(L,search(K,V),L1), Cs=nt(K1,V1,L1,R).
cons(C,search(K,V),Cs) :- C=nt(K1,V1,L,R), K>K1 |              %4
          cons(R,search(K,V),R1), Cs=nt(K1,V1,L,R1).
cons(C,update(K,V),Cs) :- C=nt(K ,_ ,L,R)       |              %5
          Cs=nt(K,V,L,R).
cons(C,update(K,V),Cs) :- C=nt(K1,V1,L,R), K<K1 |              %6
          cons(L,update(K,V),L1), Cs=nt(K1,V1,L1,R).
cons(C,update(K,V),Cs) :- C=nt(K1,V1,L,R), K>K1 |              %7
          cons(R,update(K,V),R1), Cs=nt(K1,V1,L,R1).
cons(C,search(_,V),Cs) :- C=t                   |              %9
          V=undefined,Cs=t.
cons(C,update(K,V),Cs) :- C=t                   |              %10
          Cs=nt(K,V,L,R),L=t,R=t.
```

Figure 7: Binary Tree Search Program (Transformed)
(functor tree, list stream notation)

**Measured Results:** All `update` and `search` messages are generated by the message generator. It sends, to the binary tree, 5000 `update` messages with randomly generated keys. It then sends 5000 `search` messages, having the same key sequence as the update messages.

Table 1 shows the measured timings and suspension counts. The parenthesized number is the number of suspensions and their ratios, as measured for the KLIC system. In this case, a programmer would probably assume

| Input Stream Type | KLIC system (Ver.1.010) Original | Dual Prog. | Speedup | PIM/p system Original | Dual Prog. | Speedup |
|---|---|---|---|---|---|---|
| | No-wait generator | | | | | |
| List stream | 299 ms | 421 ms | 0.71 | 1834 ms | 1823 ms | 1.01 |
| | (0) | (0) | – | | | |
| Functor stream | 267 ms | 306 ms | 0.87 | 1623 ms | 1511 ms | 1.07 |
| | (0) | (0) | – | | | |
| | Acknowledge-wait generator | | | | | |
| List stream | 957 ms | 421 ms | 2.25 | 2930 ms | 1802 ms | 1.63 |
| | (103986) | (0) | – | | | |
| Functor stream | 965 ms | 309 ms | 3.12 | 2826 ms | 1489 ms | 1.90 |
| | (103986) | (0) | – | | | |

Table 1: Performance of Binary Tree Search Program

that the generator and binary-tree processes operate like coroutines; each message is generated, then consumed, one after the other. But, since the no-wait generator generates all the messages before the binary-tree processes accept the first message, no suspension occurs. Therefore, the performance of both programs deviates only slightly (for the PIM/p system) or degraded about 30 % (for the KLIC system). The main reason of this performance degradation in the KLIC system is the execution overhead of functor processing in the KLIC system. Without suitable process-priority control, many generators in actual programs may run as no-wait generators. If the list stream of the original program is converted to a functor stream, the performance of the both systems improves.

For the acknowledge-wait generator, the performance of the original program is degraded significantly, since every `search` message incurs a suspension/resumption overhead. However, the transformed program is never suspended at all. As a result, the performance is up to 3.1 times faster on the KLIC system, and twice as fast on the PIM/p system.

### 4.1.2 Life-Game Program

**Program Description:** Figure 8 shows the kernel part of a life-game simulation program. In this program, the processes construct a mesh network where each node process is connected to its upper, lower, left- and right-hand neighbors. Each node process awaits messages from its neighbors. Once a node receives four messages, i.e. one from each neighbor, the node sends a new message back to those neighbors, then enters a new generation. Therefore, no node process can continuously run for two reductions without suspension. In an actual life-game program, each node is connected to eight neighbors and also computes the next state, but this example omits the state calculation for simplification. Figure 8 shows the kernel part of the life-game program. The transformed program is shown in Figure 9.

```
node(Li,Lo,Ri,Ro,Ui,Uo,Di,Do) :- Li=cons(T,Lis),
                        Ri=[T|Ris],Ui=[T|Uis],Di=[T|Dis],T>0,T1:=T-1 |
        Lo=[T1|Los], Ro=cons(T1,Ros), Uo=[T1|Uos], Do=[T1|Dos],
        node(Lis,Los,Ris,Ros,Uis,Uos,Dis,Dos).
node(Li,Lo,Ri,Ro,Ui,Uo,Di,Do) :- Li=cons(T,Lis),
                        Ri=[T|Ris],Ui=[T|Uis],Di=[T|Dis],T=:=0       |
        Lo=[], Ro=nil, Uo=[],Do=[],
        node(Lis,Ris,Uis,Dis).
node(Li,Ri,Ui,Di) :- Li=nil, Ri=[],Ui=[],Di=[]                       | true.
```

Figure 8: Kernel Part of the Life-Game Program (Original)

```
cons(Li,T,Lis) :- Li=node(Lo,Ri,Ro,Ui,Uo,Di,Do),
                        Ri=[T|Ris],Ui=[T|Uis],Di=[T|Dis],T>0,T1:=T-1 |
        Lo=[T1|Los], cons(Ro,T1,Ros), Uo=[T1|Uos],Do=[T1|Dos],
        Lis=node(Los,Ris,Ros,Uis,Uos,Dis,Dos).
cons(Li,T,Lis) :- Li=node(Lo,Ri,Ro,Ui,Uo,Di,Do),
                        Ri=[T|Ris],Ui=[T|Uis],Di=[T|Dis],T=:=0       |
        Lo=[], nil(Ro), Uo=[], Do=[],
        Lis=node(Ris,Uis,Dis).
nil(Li) :- Li=node(Ri,Ui,Di), Ri=[],Ui=[],Di=[]                      | true.
```

Figure 9: Kernel Part of the Life-Game Program (Transformed)

**Duality Transformation:** Compared to the previous example, a dual program is hard to conceive and write from scratch. To transform it into a dual program, although each node actually has four input streams, here, we treat only `Li` (the input from the left neighbor) as the input stream. Therefore, `Ro` (the output to the right neighbor) should be treated as the output stream and the rest of the arguments as a process state as a whole. Since this transformation does not change the substantial algorithm, the transformed program does not deadlock, in the same way as the original.

**Measured Results:** We created a small mesh network ($10 \times 10$) and a large mesh network ($20 \times 20$), and executed 1000 generations. The measured results (Table 2) show that, in both systems, the transformed programs are always faster than the originals. As in the previous table, the figures within parentheses are those for suspension. As we mentioned before, each `node` predicate in the original program is suspended at least every reduction and, worse still, two or more suspensions may occur during one reduction. The behavior of a transformed program may be difficult to understand, but it can continue to run for a certain number of reductions without suspension. Consequently, the thread is prolonged 2.9 to 5.2 times of the original one, and the performance is improved $1.3 \sim 1.9$ times.

| Mesh Size | KLIC system (Ver.1.010) | | | PIM/p system | | |
|---|---|---|---|---|---|---|
| | Original | Dual Prog. | Speedup | Original | Dual Prog. | Speedup |
| $10 \times 10$ | 1040 ms | 600 ms | 1.73 | 3009 ms | 2334 ms | 1.30 |
| | (220858) | (76982) | (2.87) | | | |
| $20 \times 20$ | 4580 ms | 2390 ms | 1.92 | 13843 ms | 9415 ms | 1.47 |
| | (845516) | (162452) | (5.20) | | | |

Table 2: Performance of Life-Game Program (1000 Generations)

### 4.2 New Programming Methodologies Based on Duality

We have seen how processes in a source program correspond to messages in its dual program. And, processes in a source program are treated as first-class objects in its dual program. Therefore, our transformation may provide a means for meta-control in concurrent logic languages. This section presents how to describe continuation based on duality in FGHC, as an example of meta-control. Further, we have prototyped the migration of FGHC processes based on the notion of the process-message duality. However, because of space limitations, we must skip the subject of migration.

**Continuation:** Consider the sample program shown in Figure 10. This program searches for a given value in a binary tree in a depth-first manner. The result from the left tree is provided to the right tree via the third argument of `search` and the first argument of `search1` (`CPs`). `search1` relays the message `found` or `not_found`. The result determines whether the program has located the right tree. Note that the data structure bound to `CPs` is a process stream whose length is equal to the depth of the tree. Since this program is a restricted FGHC program, we can transform the program with respect to the variables `CP` and `CPs` (Figure 11). Then, after the

```
?- search(Key,Tree,Res), cont(Res,Args).

search(K,T,CP):- T=leaf(Ts), K = Ts  | CP = found.
search(K,T,CP):- T=leaf(Ts), K \= Ts | CP = not_found.
search(K,T,CP):- T=node(T0,T1)       |
        search(K,T0,CPs), search1(CPs,K,T1,CP).

search1(CP,K,T1,CPs):- CP = found     | CPs = found.
search1(CP,K,T1,CPs):- CP = not_found | search(K,T1,CPs).

cont(CP,Args):- CP=found     | found_proc(Args).
cont(CP,Args):- CP=not_found | not_found_proc(Args).
```

Figure 10: Tree Search in FGHC

```
?-search(Key,T,Res), Res=cont(Args).

search(K,T,CP):- T=leaf(Ts), Ts = K  | found(CP).
search(K,T,CP):- T=leaf(Ts), Ts \= K | not_found(CP).
search(K,T,CP):- T=node(T0,T1)       |
        search(K,T0,CPs), CPs=search1(K,T1,CP).

found(CP):- CP=search1(K,T1,CPs) | found(CPs).
found(CP):- CP=cont(Args)        | found_proc(Args).

not_found(CP):- CP=search1(K,T1,CPs) | search(K,T1,CPs).
not_found(CP):- CP=cont(Args)        | not_found_proc(Args).
```

Figure 11: Dual Tree Search Program

transformation, each clause has a maximum of one body predicate. Hence, we may say that the transformed program presents its intrinsic sequentiality more explicitly, and runs efficiently from an implementation point of view. In Figure 11, the data structure occurring at the third argument of `search` has the following form (called `search1`-chain):

```
CP=search1(K,T1,CP'),CP'=search1(K,T1',CP''),...,CP'⋯'=cont(Args)
```

The function symbol `search1` can be regarded as being the future execution after searching the left tree, and the chain of the terms, `search1`-chain, in a heap explicitly designates the possible execution order. Here, we say that the transformed program is of the continuation-passing style (CPS).

When a leaf whose value is equal to a given key is found, the predicate `found` is invoked, and traverses the `search1`-chain. Finally, the traverse reaches `cont`, after which control is passed to the next computation `found_proc`. However, considering the behavior of this program, it is not efficient to traverse every `search1` of the chain once the target leaf has been found. Thus, more efficient execution can be realized by invoking the `found_proc` predicate directly (Figure 12). In the program shown in Figure 12, the number of arguments of `search` is increased from three to four; the extra argument holds the continuation to be executed immediately upon `found` is being invoked. By performing a transformation with respect to the

```
?-search(Key,T,Res,Res), Res=cont(Args).

search(K,T,CP,FC):- T=leaf(Ts), Ts = K  | found(FC).
search(K,T,CP,FC):- T=leaf(Ts), Ts \= K | not_found(CP,FC).
search(K,T,CP,FC):- T=node(T0,T1)       |
         search(K,T0,CPs,FC), CPs=search1(K,T1,CP).

found(FC):- FC=cont(Args) | found_proc(Args).

not_found(CP,FC):- CP=search1(K,T1,CPs) | search(K,T1,CPs,FC).
not_found(CP,FC):- CP=cont(Args)        | not_found_proc(Args).
```

Figure 12: More Efficient Dual Tree Search

variables to derive a computation result, we obtain a program where the execution is represented by first-class objects.

Consider the program obtained by the duality-transformation of the program in Figure 12. Prior to the transformation, the goal in the figure must be normalized by the duplication method, as follows:

```
?-search(Key,T,Res1,Res2),Res1=cont(Args),Res2=cont(Args).
```

Subsequently the transformation with respect to `Res1` and `Res2` seems applicable. However, as you have probably noticed, the normalized program does not satisfy restriction **(5)** in Section3.2, since the normalized program discards the `search1`-chain without reading it. If we ignore restriction **(5)**, unprocessed data (garbage data) in the program will be transformed to unexecuted processes (deadlock processes) from the duality point of view.

# 5 Conclusion

We have proposed a new transformation method for concurrent logic languages. The method enables transformation between message-oriented programs and process-oriented programs. By examining predicate invocation and message sending/receiving processes in a concurrent logic language, we have clarified the notion of the duality of processes and messages. Our method does not change the algorithm of a program, instead changing the data carriers (process to message and vice versa). With our method, a programmer can write programs as he or she desires, then transform them to more efficient (dual) ones. That is, to some extent, our method can free programmers from considering their programs' dynamic behavior and performance. We have evaluated the performance improvement gained by the use of our method on two systems, the KLIC system and the PIM/p system, both of which were developed at ICOT. For the benchmark programs referenced in this paper, the duality transformation achieves a threefold maximum speedup.

Furthermore, our method not only increases the choices available during programming and improves performance, but also enables us to express control as a first-class object. Hence, meta-programming based on duality is becomes possible. As an example, continuation can be described in FGHC without meta-language facilities.

Consequently, the duality of messages and processes in concurrent logic languages enriches our programming techniques and methodologies, and also contributes to performance improvement.

## Acknowledgments

The authors thank Kazunori Ueda for his valuable advice. The authors are also grateful to the anonymous referees of a previously submitted version of this paper for their useful suggestions. Mention is also due to our ICOT researchers, and the stimulating research environment they generate.

## References

[1] T. Chikayama *et al.* A Portable and Reasonably Efficient Implementation of KL1. In *Proc. of the Tenth International Conference on Logic Programming*, page 833. The MIT Press, 1993.

[2] J. Crammond. The Abstract Machine and Implementation of Parallel Parlog. *New Generation Computing*, 10(4):385–422, 1992.

[3] S. K. Debray. Static Analysis of Logic Programs. In *An Advanced Tutorial of the ILPS'93*, 1993.

[4] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, 1989.

[5] D. Gudeman, K. De Bosschere, and S. K. Debray. `jc`: An Efficient and Portable Sequential Implementation of Janus. In *Proc. of ICLP'92*, 1992.

[6] K. Hirata *et al.* Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1. In *Proc. of International Conference on Fifth Generation Computer Systems*, pages 436–459. ICOT, 1992.

[7] K. Kumon *et al.* Architecture and Implementation of PIM/p. In *Proc. of International Conference on Fifth Generation Computer Systems*, pages 414–424. ICOT, 1992.

[8] E. Shapiro and A. Takeuchi. Object-oriented programming in Concurrent Prolog. *New Generation Computing*, 1(1):25–49, 1983.

[9] A. Takeuchi. *Parallel Logic Programming.* John Wiley & Sons, Inc., 1992.

[10] E. Tick and C. Banerjee. Performance Evaluation of Monaco Compiler and Runtime Kernel. In *Proc. of ICLP'93*, 1993.

[11] K. Ueda and M. Morita. Message-Oriented Parallel Implementation of Moded Flat GHC. In *Proc. of International Conference on Fifth Generation Computer Systems*, pages 799–807. ICOT, 1992.

[12] R. Yang *et al.* Performance of the Compiler-based Andorra-I System. In *Proc. of ICLP'93*, 1993.