ICOT Technical Memorandum: TM-0141

TM-0141

GHC Process Fusion by Program Transformation

by
K. Furukawa and K. Ueda

October, 1985

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

GHC PROCESS FUSION BY PROGRAM TRANSFORMATION

Koichi Furukawa, Kazunori Ueda

Institute For New Generation Computer Technology

A program transformation method for GHC process fusion is presented. It is based on the fold/unfold method by Burstall and Darlington. The method is applied to a simple logic circuit simulation program. Furthermore, the importance of the process fusion is discussed in the context of knowledge representation as well as of modular programming.

1. Introduction

In a stream-oriented computation formalism, computation activities are distributed among processes possibly communicating with one another. In general, each process is preferably designed to compute a relatively simple and small task in order to keep it understandable. But the resulting programs may generate too many tiny processes, which, if naively implemented, will cause extra computation for interprocess communication. Process fusion is aimed at reducing the number of processes by fusing some communicating processes, and this is analogous to loop fusion in procedural languages. Such fusion may not necessarily improve time efficiency: Process fusion might decrease efficiency in a highly parallel architecture, and loop fusion might prevent a compiler to "vectorize" loops. However, an actual parallel processing system may well have a fairly large granularity, and process fusion can be used for adapting the grain size of each process to the granularity of the underlying system.

We use GHC (Guarded Horn Clauses) [Ueda 85] to express our programs. In fact, we are interested only in the subclass of GHC which allows no user-defined goals in guards. Our approach to the process fusion is based on the program transformation method by folding/unfolding first proposed in [Burstall and Darlington 77] and applied to logic programming in [Tamaki and Sato 84]. However, since GHC has introduced a causality concept into Horn clauses, the correct transformation must consider causality also. Our transformation can be formally justified only on the semantics of GHC which describes its causality concept, and that semantics will be obtained by extending the formal semantics of nondeterministic dataflow languages. In this paper, however, we justify our transformation by informal arguments.

In the following, we will informally state a method for process fusion using a simple example. The method is then applied to a simple logic circuit simulation program simulating a series of two inverters to obtain a program simulating a simple delay circuit. Furthermore, the importance of the process fusion is discussed in the context of knowledge representation as well as of modular programming.

2. Sketch of the Process Fusion Method

In this section, we will briefly sketch the process fusion method using a simple example of computing a sequence of partial sums of integer sequences. The original GHC program is as follows:

```
integerSums(I,N,Sums) :- true |
  integers(I,N,Is), sums(Is,Sums).          (1)

integers(I,N,Is) :- I=<N |
  J := I+1, Is=[I|Its],
  integers(J,N,Its).                        (2)
integers(I,N,Is) :- I>N | Is=[].            (3)

sums(Is,Sums) :- true | sum1s(Is,0,Sums).   (4)

sum1s([],_,Sum1s) :- true | Sum1s=[].       (5)
sum1s([I|Is],Temp,Sums) :- true |
  NewTemp := I+Temp,
  SUMs=[NewTemp|Sum1s],
  sum1s(Is,NewTemp,Sum1s).                  (6)
```

Two tail-recursive goals 'integers' and 'sum1s' will appear in the computation, which are regarded as processes here. The other goals could also be called processes, but here we do not regard them as processes since they are all short-lived. Our objective is to obtain a one-process program which computes the same sequence. We start from Clause (1) consisting of two goals 'integers' and 'sums', and unfold or 'execute' these body goals until we have the two tail-recursive goals:

```
integerSums(I,N,Sums) :- true |
  integers(I,N,Is), sum1s(Is,0,Sums).       (7)
```

The correctness of this transformation should be clear: The predicate 'sums' has only one clause with an empty guard, so Clause (4) is always selected.

Then we introduce a new clause for the final single process by parameterizing the second argument of 'sum1s' and leaving the shared variable 'Is' for communication local. This is the key step which usually needs heuristics in order to obtain a proper clause to accomplish the process fusion. In our case, however, the only heuristics we need is to generalize parameters and this can easily be automated. Let us name the new predicate 'fused_integerSums'.

The resulting clause is as follows:

```
fused_integerSums(I,N,Temp,Sums) :- true |
  integers(I,N,Is), sum1s(Is,Temp,Sums).     (8)
```

The second argument of 'sum1s' is generalized from a constant 0 to a variable Temp, and it is included in the clause head. Now we try to transform this clause to get a single tail-recursive program. To obtain the expected result, we want to apply unfolding and then folding. However, the goal 'sum1s' can be expanded into its body goals only when 'integers' has instantiated 'Is', and 'integers' can be expanded only when the value of 'I' and 'N' and their ordering is known. This means that Clause (8) is allowed to wait for and compare the values of 'I' and 'N' on behalf of 'integers'. So we split Clause (8) into two:

```
fused_integerSums(I,N,Temp,Sums) :- I=<N |
  integers(I,N,Is), sum1s(Is,Temp,Sums).     (9)
fused_integerSums(I,N,Temp,Sums) :- I>N |
  integers(I,N,Is), sum1s(Is,Temp,Sums).    (10)
```

The bodies of Clause (9) and (10) can assume that I=<N and I>N, respectively, and now we can unfold 'integers' and then 'sum1s' without worrying about their guards:

```
fused_integerSums(I,N,Temp,Sums) :- I=<N |
  J is I+1, Is = [I|I1s],
  integers(J,N,I1s),
/* sum1s([I|I1s],Temp,Sums) */
  NewTemp is I+Temp,
  Sums = [NewTemp|Sum1s],
  sum1s(I1s,NewTemp,Sum1s).                  (11)
fused_integerSums(I,N,Temp,Sums) :- I>N |
  Is=[],
/* sum1s([],Temp,Sums) */
  Sum1s=[].                                  (12)
```

By folding 'integers' and 'sum1s' by Clause (8) and forgetting the intermediate variable 'Is', we obtain the following:

```
fused_integerSums(I,N,Temp,Sums) :- I=<N |
   J is I+1, NewTemp is I+Temp,
   Sums = [NewTemp|Sum1s],
   fused_integerSums(J,N,NewTemp,Sum1s).     (13)
fused_integerSums(I,N,Temp,Sums) :- I>N |
   Sum1s=[].                                 (14)
```

Folding by Clause (8) has no problem, since it has an empty guard and the shared variable 'I1s' in Clause (11) is local to the two goals to be folded.

The remaining task is to express the original predicate 'integerSums' in terms of the newly introduced predicate 'fused_integerSums'. This can be easily done by setting the parameter introduced in Clause (8) to the original value:

```
integerSums(I,N,Sums) :- true |
   fused_integerSums(I,N,0,Sums).            (15)
```

The resulting clauses (13), (14) and (15) give the new definition of the 'integerSums' program. This program contains only one tail-recursive process; thus we have fused the original two processes into one. The intermediate stream 'Is' has been eliminated together with the operations of composing and decomposing it. So the new program should be more efficient with respect to any naive measure which counts every primitive operations separately. From a practical point of view, the compiled code for a sequential machine obtained from the new program should be better than the code obtained by compiling the original two tail-recursive procedures separately.

3. More on Unfolding/Folding Transformation

In this section, we will show how we can deal with a stream transformer which may absorb some of the input elements. Such a transformer cannot be handled in the framework of [Wadler 81].

```
evenSquare(Xs,Ys) :- true |
  evenseq(Xs,Es), squareseq(Es,Ss).          (16)
evenseq([X|Xs],Ys) :- even(X) |
  Ys=[X|Y1s], evenseq(Xs,Y1s).               (17)
evenseq([X|Xs],Ys) :- odd(X) |
  evenseq(Xs,Ys).                            (18)

squareseq([U|Us],Vs) :- true | X:=U^2,
  V=[X|V1s], squareseq(Us,V1s).              (19)
```

In this example, the two processes are asynchronous in nature. The 'evenseq' process filters out odd numbers from the input sequence Xs by (18), so it does not synchronize with the 'squareseq' process in general. Anyway, let us first split Clause (16), as we did in the 'integerSums' example.

```
evenSquare([X|Xs],Ys) :- even(X) |
  evenseq([X|Xs],Es), squareseq(Es,Ss).      (20)
evenSquare([X|Xs],Ys) :- odd(X) |
  evenseq([X|Xs],Es), squareseq(Es,Ss).      (21)
```

When the head element 'X' is even, it is passed to the 'squareseq' process. Therefore, we unfold both of the body goals of Clause (20) and get the following clause:

```
evenSquare([X|Xs],Vs) :- even(X) |
  Y:=X^2, Vs=[Y|V1s], evenSquare(Xs,V1s).    (22)
```

On the other hand, when the head element is odd, no output is generated from the 'evenseq' process. So we must unfold only the 'evenseq' process to get a foldable pattern. The unfolding/folding steps are shown below:

```
evenSquare([X|Xs],Ys) :- odd(X) |
  evenseq([X|Xs],Es), squareseq(Es,Ss).      (21)
            |
            |  unfold 'evenseq'
            V
evenSquare([X|Xs],Vs) :- odd(X) |
  evenseq(Xs,Ys), squareseq(Ys,Vs).          (23)
            |
            |  fold by Clause (16)
            V
evenSquare([X|Xs],Vs) :- odd(X) |
  evenSquare(Xs,Vs).                         (24)
```

The same technique can be applied to the following list compaction program to reduce the number of 'remove' processes.

```
compact([],X) :- true | X=[].                          (25)
compact([H|T],Z1) :- true |
  Z1=[H|Z], remove(H,T,T1),
  compact(T1,Z).                                       (26)

remove(H,[],X) :- true | X=[].                         (27)
remove(H,[H|T],U) :- true | remove(H,T,U).             (28)
remove(H,[A|T],U) :- H=\=A |
  U=[A|V], remove(H,T,V).                              (29)
```

We introduce a new predicate 'doubleRemove'

```
doubleRemove(H1,H2,T1,T3) :- true |
  remove(H1,T1,T2), remove(H2,T2,T3).                  (30)
```

and fuse every adjacent pair of 'remove' processes. The resulting program is as follows:

```
compact([],Z1) :- true | Z1=[].                        (31)
compact([H],Z1) :- true | Z1=[H].                      (32)
compact([H1,H2|T],Z1) :- H1=\=H2 |
  Z1=[H1,H2|Z],
  doubleRemove(H1,H2,T,V1), compact(V1,Z).             (33)
compact([H1,H1|T],Z1) :- true |
  compact([H1|T],Z1).                                  (34)

doubleRemove(H1,H2,[],V1) :- true | V1=[].             (35)
doubleRemove(H1,H2,[H1|T],V1) :- true |
  doubleRemove(H1,H2,T,V1).                            (36)
doubleRemove(H1,H2,[H2|T],V1) :-
  H1=\=H2 |
  doubleRemove(H1,H2,T,V1).                            (37)
doubleRemove(H1,H2,[H3|T],V1) :-
  H1=\=H3, H2=\=H3 | V1=[H3|V2],
  doubleRemove(H1,H2,T,V2).                            (38)
```

The 'H1=\=H2' check in Clause (37) can further be omitted because this condition is guaranteed to hold by Clause (33). Even without Clause (33), this check can be omitted because Clause (37) can handle the 'H1=H2' case correctly. Note that the introduction of 'doubleRemove' is rather arbitrary in that we could fuse more than two 'remove' processes by introducing, say, 'tripleRemove'.

4. An Example of Logic Simulation

In this section, we apply our technique to a logic simulation program. The example program simulates an circuit of two inverters connected in series. The aim of the fusion is to transform the program into a simple program with only one delay component. The original definition of the two-inverter program is as follows.

```
doubleInverter(T,I,O) :- true |
   inverter(T,I,O1), inverter(T,O1,O).

inverter(T,I,O) :- true |
   gate1inInit(inv,T,I,O).

gate1inInit(Type,T,I1,Out) :- true |
   delayTime(Type,D),
   initialOut(D,Out,Os),
   gate1in(Type,T,I1,Os).

initialOut(D,X,Os) :- D>0 |
   X=[x|O1s], D1 := D-1,
   initialOut(D1,O1s,Os).
initialOut(0,Os,X) :- true | X=Os.

gate1in(_,[],_,O) :- true | O=[].
gate1in(Type,[_|Ts],[I1|I1s],Outs) :- true |
   Outs=[Ev|Out1s],
   truthValue(Type,I1,Ev),
   gate1in(Type,Ts,I1s,Out1s).

delayTime(inv,X) :-  true | X = 1.

truthValue(inv,1,X) :- true | X = 0.
truthValue(inv,0,X) :- true | X = 1.
truthValue(inv,x,X) :- true | X = x.
```

The essential part of the above program is the 'gate1in(Type,T,I,O)' procedure: a tail-recursive procedure for a component of the type 'Type' with one input 'I' and one output 'O'. The second argument 'T' represents a sequence of clock signals. The 'initialOut' procedure brings the delay of the inverter into the output stream 'Os'. The fusion process is illustrated in the following steps.

Step 1. Unfold the 'doubleInverter' until obtaining two tail recursive goals.

```
doubleInverter(T,I,O) :- true |
  delayTime(inv,D1), initialOut(D1,O1,O1s),
  gate1in(inv,T,I,O1s),
  delayTime(inv,D2), initialOut(D2,O,O2s),
  gate1in(inv,T,O1,O2s).
                |
                | unfold (execute) 'delayTime'
                | and 'initialOut'
                V
doubleInverter(T,I,O) :- true |
  gate1in(inv,T,I,O1s),
  Os=[x|O2s],
  gate1in(inv,T,[x|O1s],O2s).
```

Step 2. Unfold further to make the output of the first 'gate1in' be equal to the input of the second 'gate1in'.

```
                |
                | case-split and
                | unfold the second 'gate1in'
                V
doubleInverter([_|Ts],Is,Os) :- true |
  gate1in(inv,[_|Ts],Is,O1s),
  Os=[x|O2s],
/*gate1in(inv,[Tc|Ts],[x|O1s],O2s)*/
  O2s=[Ev|Outs],
  truthValue(inv,x,Ev),
  gate1in(inv,Ts,O1s,Outs).
                |
                | execute 'truthValue'
                V
doubleInverter([_|Ts],Is,Os) :- true |
  gate1in(inv,[_|Ts],Is,O1s),
  Os=[x,x|Outs],
  gate1in(inv,Ts,O1s,Outs).
```

Step 3. Define a folding predicate by generalizing 'doubleInverter'.

```
fusedDoubleInverter(Ts,Is,Outs) :- true |
  gate1in(inv,[_|Ts],Is,O1s),
  gate1in(inv,Ts,O1s,Outs).
```

Step 4. Express 'doubleInverter' in terms of 'fusedDoubleInverter' by folding.

```
doubleInverter([_|Ts],Is,Os) :- true |
  Os=[x,x|Outs],
```

```
  fusedDoubleInverter(Ts,Is,Outs).
```

Step 5. Transform the 'fusedDoubleInverter' into a tail recursive program.

```
                |
                | case-split and
                | unfold two 'gate1in's
                V
fusedDoubleInverter([_|Ts],[I|Is],Outs) :-
  true |
/*gate1in(inv,[_,_|Ts],[I|Is],O1s)*/
  O1s=[Ev|Out1s],
  truthValue(inv,I,Ev),
  gate1in(inv,[_|Ts],Is,Out1s),
/*gate1in(inv,[_|Ts],O1s,Outs)
  where O1s=[Ev|Out1s]*/
  Outs=[Ev2|Out2s],
  truthValue(inv,Ev,Ev2),
  gate1in(inv,Ts,Out1s,Out2s).
                |
                | fold by itself
                V
fusedDoubleInverter([_|Ts],[I|Is],Outs) :-
  true |
  truthValue(inv,I,Ev1),
  Outs=[Ev2|Out2s],
  truthValue(inv,Ev1,Ev2),
  fusedDoubleInverter(Ts,Is,Out2s).
```

By further unfolding the above two 'truthValues', it is easily shown that the 'fusedDoubleInverter' becomes a simple two-unit delay circuit.

It is possible to generalize the 'fusedDoubleInverter' by changing the constant 'inv' into variables. The resulting procedure would correspond to a general cascade of two components with one input. We also succeeded in transforming a flip-flop represented by two NAND gates into a single process procedure. In this case, two 'gate2in' procedures were fused instead of two 'gate1in's in the above example.

5. Concluding Remarks

We have demonstrated how to apply program transformation technique to achieve GHC process fusion. In this section, we show the importance of stream-oriented programming and process fusion with respect to knowledge representation and programming methodology.

As the logic simulator example shows, a GHC process is appropriate for representing a dynamic object. Furthermore, GHC allows one to define a larger process by means of subprocesses. For example, 'integerSums' is a compound object with an integer generator and a summation machine; the 'doubleInverter' is a compound object with two inverters connected serially. Thus a GHC clause can represent part-whole relationship.

Part-whole relationship is one of the central issues for knowledge representation. The difficult problem is how to capture the whole as its own existence. Since a whole consists of a set of parts, it is natural to think that atomic parts are the only things that actually exist and the whole is only a conceptual thing. A GHC program behaves just like this; when the process representing a whole is called, it is repeatedly replaced by the processes representing its parts until the only active processes are tail-recursive parts.

Process fusion in this context means to rebuild a whole system by means of short-lived primitives and single tail-recursion to reduce overhead caused by using higher-level standard parts. This method will increase the efficiency of the system without changing its function. Process fusion is a kind of meta-level activities in the sense that it manipulates program definitions themselves. Therefore it is natural to be integrated into knowledge representation/programming systems such as Mandala [Furukawa et al. 84].

From the viewpoint of programming methodology, stream-oriented programming encourages modular programming. That is, stream-oriented programming allows us to construct a program by connecting its part processes by shared variables. In the integer summation example, the entire program consists of two parts, 'integers' and 'sums'. Each of them is a general part which can be used also in other programs. If the program were written as a single process from the first, it would have less reusability. Process fusion will encourage modular programming based on the reuse of parts, since it removes possible inefficiency caused by the extensive use of small parts.

As for the process fusion mechanism, one research direction is to investigate process structures and to classify the pattern of the possible transformation in terms of them. This will clarify the abilities and limits of our method.

Acknowledgments: We acknowledge Akikazu Takeuchi, Jiro Tanaka, and the other researchers of the first laboratory of ICOT Research Center for their helpful suggestions and comments. We also acknowledge Yasunori Noda, Tetsuo Kinoshita, Akira Okumura and others from Oki Electric Company who developed a parallel logic simulator. Their programs helped very much to examine our ideas. We also thank to Kazuhiro Fuchi, the Director of ICOT Research Center, for providing the opportunity to conduct this research and also for encouraging us.

References

[Burstall and Darlington 77] A Transformation System for developing Recursive Programs, JACM Vol.24, No.1.

[Furukawa et al. 84] Mandala: A Logic Based Knowledge Programming System, Proc. FGCS '84.

[Tamaki and Sato 84] Unfold/Fold Transformation of Logic Programs, Proc. 2nd Int. Logic Programming Conf., Uppsala.

[Ueda 85] Guarded Horn Clauses, ICOT Tech. Report TR-103.

[Wadler 81] Applicative Style Programming, Program Transformation, and List Operators, Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture, ACM.