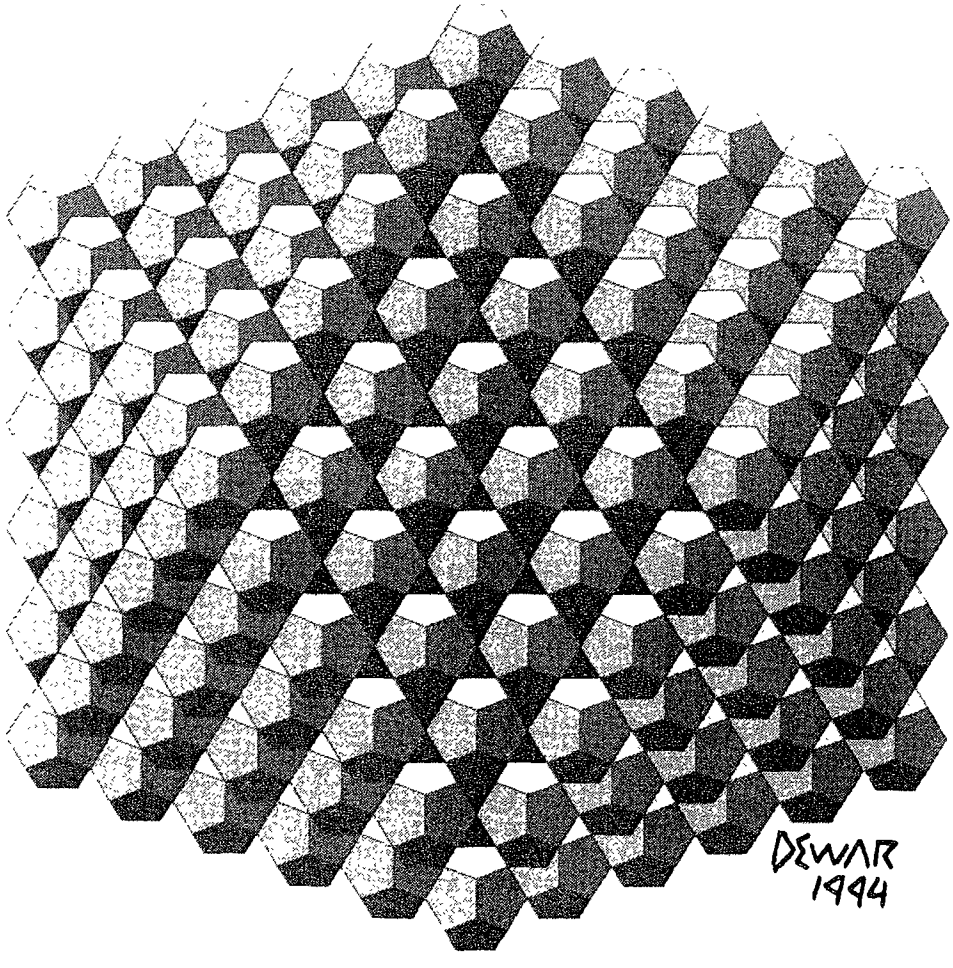
Symmetry: Culture and Science

Symmetry:
Natural and Artificial, 2

The Quarterly of the
International Society for the
Interdisciplinary Study of Symmetry
(ISIS-Symmetry)

Editors:
György Darvas and Dénes Nagy

Volume 6, Number 2, 1995



Third Interdisciplinary Symmetry Congress and Exhibition
Washington, D.C., U.S.A. August 14 - 20, 1995

SYMMETRICAL PROGRAMS AND RÔLE SWITCHING

Prof. W. Douglas Maurer
Department of Electrical Engineering and Computer Science
The George Washington University
Washington, D. C. 20052
Email: maurer@seas.gwu.edu

The term "rôle switching" is derived from a story that this author once read about two very experienced actors, touring in a play (I think it was *The Odd Couple*, but I might be wrong) involving rôles for two men of about the same age. Every so often, according to the story, the actors would switch rôles, just as a diversion; Oscar would be Felix for one night, and Felix would be Oscar.

For some time we have been studying rôle switching, an advanced programming technique of general utility, in connection with the extension to computer science of what is commonly known in mathematics as proof by symmetry. Suppose for example that we have the following two definitions (adapted from this author's paper [1]):

(1) Every A is either a D or it is another A followed by a D.

(2) Every B is either a D or it is another B preceded by a D.

We wish to prove that every A is a B, and vice versa. Intuitively it is clear that every A is just a sequence of D's, and every B is the same, but suppose that we want to apply formal induction. Then we might proceed as follows:

PART 1 (every A is a B). Let X be an A. If X is a D, then it is a B. If X is Y followed by a D, where Y is another A, then Y is a B, by induction. Hence it suffices to prove that a B followed by a D is always a B. Let U be a B. If U is a D, then U, followed by a D, is a D followed by a D. This is a B because it is a D preceded by another D, which is also a B. If U is V preceded by a D, where V is a B, then consider U followed by a D. This is a D followed by V followed by another D. By induction, V followed by a D is a B. Therefore we have a D followed by a B, or a B preceded by a D, which is a B, by definition.

PART 2 (every B is an A). Let X be a B. If X is a D, then it is an A. If X is Y preceded by a D, where Y is another B, then Y is an A by induction. Hence it suffices to prove that an A preceded by a D is always an A. Let U be an A. If U is a D, then U, preceded by a D, is a D preceded by a D. This is an A because it is a D followed by another D, which is also an A. If U is V followed by a D, where V is an A, then consider U preceded by a D.

This is a D preceded by V preceded by another D. By induction, V preceded by a D is an A. Therefore we have a D preceded by an A or an A followed by a D, which is an A, by definition.

This proof contains considerable redundancy. Indeed, if we replace "an A" by "a B" and "a B" by "an A," throughout the first paragraph, and similarly "preceded" by "followed" and "followed" by "preceded," we get precisely the second paragraph. Customarily, in mathematics, we do not write out the second paragraph in a case like this, but merely write, instead of it:

PART 2 (every B is an A). This is true by symmetry.

In computer science we often encounter a similar situation, not with two parts of a proof, but with two parts of a program; the first can be transformed into the second by replacing A by B and B by A (or the like). The classical way of treating this situation is to construct a procedure P with two parameters. The first part of the program is invoked by calling P(A, B); the second part is invoked by calling P(B, A).

As it stands, this has several disadvantages. There is always some function call overhead; there is always some parameter-passing overhead; and, since A and B might be changed, they have to be passed as pointers, with further overhead when we change what a pointer points to. Rôle switching avoids all of these problems.

In rôle switching, we set up two registers, say R1 and R2, which contain the values of A and B respectively. We say that R1 plays the rôle of A, while R2 plays the rôle of B. We now write the first part of our program in terms of R1 and R2, rather than A and B. When we are ready to do the second part of our program, we simply exchange the numbers in the registers A and B. (Some computers will even let you do that in one machine language instruction.) Now R1 plays the rôle of A, while R2 plays the rôle of B; the rôles have been switched, and we can merely jump to the first part of the program.

We now give three specific examples of rôle switching. The first involves a linked list merge program which we presented, for 68000 assembly language, in [2]; the algorithm has been well known for some time. We merge two linked lists L1 and L2 into a new list L3 by manipulating pointers, leaving the list nodes where they are. At any point in the program, we know where the end of L3 is, and we are comparing the elements H1 and H2 at the heads of L1 and L2. There are two cases:

Case 1: H1 is smaller (or equal). Hook up the end of L3 to H1 and note H1 as the new end of L3. What H1 points to becomes the new H1. If the H1 pointer is now zero, then hook L3 onto H2; the rest of L3 is simply the rest of H2, and the program is finished.

Case 2: H2 is smaller (or equal). Hook up the end of L3 to H2 and note H2 as the new end of L3. What H2 points to becomes the new H2. If the H2 pointer is now zero, then hook L3 onto H1; the rest of L3 is simply the rest of H1 and the program is finished.

As in our mathematical theorem, we note that these two paragraphs are almost identical; in this case, replacing H1 by H2 and H2 by H1 transforms Case 2 into Case 1. Using rôle switching, we set two registers U and V to H1 and H2 respectively. If we are in Case 2, we exchange U and V, and simply go to the start of Case 1.

Our second example involves coroutines, also presented in [2] for 68000 assembly language. Here the basic problem is the stack-oriented call and return statements. In older computers (and also, by the way, in the newer RISC machines), a subroutine call instruction puts the return address in a register, and there is often an instruction which calls what is in some register. Coroutine transfer is then simply a matter of calling what is in register R with an instruction which also leaves the return address in register R. In this way, whenever a coroutine is executing, the return address for the other coroutine is in R, and this situation is preserved by a coroutine transfer.

This straightforward method is not usable on present-generation machines which call a procedure by pushing the return address on the stack, and return by popping an address from the stack and jumping to it. The problem is solved by means of rôle switching. We keep two stacks and two registers for them; one is the built-in stack register (A7, on the 68000) and the other is a register not used for other purposes (such as A4). This is initialized to A7, and then we reserve SMAX bytes for the A4 stack (for some stack maximum value SMAX) by decreasing A7 by SMAX (note that stacks grow downward on these machines).

In order to make a coroutine transfer from either program, we call COTRANS, which exchanges A7 and A4 and then returns in the normal manner. Note that calling COTRANS from the coroutine with the A7 stack will put the return address (call it R7) on that stack, and then switch rôles and return, so that when COTRANS is called again, A7 and A4 go back the way they were before. This means that returning, in the normal manner, will pop R7 from the stack and jump to it, which is exactly the behavior we want. The same is then true by symmetry when COTRANS is called from the coroutine with the A4 stack.

Our final example of rôle switching involves three registers, and three rôles. This program was extensively described in [3]; we reproduce the essential ideas here so as to compare them with other rôle switching applications. Suppose that we are doing a binary search, with three local variables, FIRST, LAST, and MID. As usual, we are searching a sorted array (call it T) for a particular element (call it X). At any point in the algorithm,

we are searching a subtable of T, whose first entry is T[FIRST]. The first entry not in the subtable is T[LAST]; while MID is the average of FIRST and LAST.

We proceed by comparing X with T[MID]. If X < T[MID], then X is in the first half of the subtable. This means that T[MID] is the first entry not in the subtable (we cannot have X = T[MID] if X < T[MID]), so we set LAST = MID. If X ≥ T[MID], then X is in the second half of the subtable. This means that T[MID] is the first entry in the subtable (we might have X = T[MID], although we do not check specifically for this), so we set FIRST = MID. This version of the algorithm has already been speeded up by using asymmetrical table bounds. (If T[MID] is the last entry in the subtable, rather than the first entry not in the subtable, then we need to set FIRST = MID+1 above, rather than FIRST = MID, and the addition of 1 takes extra time.)

The rôle-switching version of the binary search also avoids the time taken by setting LAST = MID and FIRST = MID. There are three registers, R1, R2, and R3, which play the rôles of FIRST, LAST, and MID. There are six different assignments of the three rôles, corresponding to the six permutations of R1, R2, and R3. Corresponding to these six assignments, there are six parts to our binary search. In each of these parts, instead of setting LAST = MID, we switch rôles; that is, we go to another part of our program, in which the rôle of FIRST remains the same, but the rôles of LAST and MID are interchanged. Similar things happen by symmetry instead of setting FIRST = MID.

As noted in [3], this algorithm can be flowcharted as a hexagon, with one of the six parts of the program at each vertex, in such a way that going from one of the six parts to another always corresponds to moving one vertex either clockwise or counterclockwise around the ring. This motivated the title of [3], honoring F. A. Kekulé, the discoverer of the benzene ring, which likewise has a symmetrical hexagonal structure.

REFERENCES

1. W. D. Maurer, "Induction principles for context-free languages," Proc. 3rd Conf. of Gesellschaft für Informatik, Hamburg, Germany, Oct. 1973, pp. 134-143.

2. W. D. Maurer, "Assembly language programming on the MAC with MPW — second draft," Memorandum GWU-IIST-90-20, June 1990.

3. W. D. Maurer, "The Kekulé algorithm: hexagonal structure in computer science," Proc. IX International Symposium "The Computer At The University," Cavtat (Dubrovnik), Yugoslavia, May 1987, pp. 11R.03 1-6.