A personal summary of Jonkers's program transformation.

(Besides being what its title says it is, this note is also a writing exercise for the right hand, which can use some training.)

The other day I served on the thesis committee for H.B.M. Jonkers; this note's main purpose is to record what seems to be the most important part of what I learned while doing so. The reason to do so publicly is to supplement EWD636 "Why naive transformation systems are unlikely to work" – written in 1977– ; Jonkers's program transformations are not necessarily "naive" in the sense with which I used the term $4\frac{1}{2}$ years ago.

Program development by means of coordinate transformation is a well-known technique, see, for instance, the fifth example in Chapter 8 of "A discipline of programming", in which an (a,c)-statespace is connected to a (p,q,r)-space by means of the equations

(0) $\qquad p = a \cdot c \quad q = c^2 \quad r = N - a^2$

The technique of relating the program operating in (a,c)-space and the program operating in (p,q,r)-space by merging them into a program operating in the Cartesian product of the two spaces is of later date and has been independently discovered by Meertens. (Jonkers gives ref. [0].) In the merged program, (0) is a set of invariant relations, and either the pair (a, c) or the triple (p, q, r) can be turned into auxiliary variables. (This exchange occurs in EWD622 "On making solutions more and more fine-grained"; in those days I still wrote about "ghost variables"!) Note that p, q, and r are not independent: they satisfy $q \cdot r = q \cdot N - p^2$.

Jonkers —see [1]— combines such a change of state space (if so desired) with a change —in particular a reduction— of a program's nondeterminacy. In aforementioned example I had changed —for the sake of efficiency— the representation without changing the degree of nondeterminacy; in many other examples I had reduced the degree of nondeterminacy without changing the state space. As a result I was puzzled to see —without any explicit justification— the two being dealt with in combination. The following considerations convinced me that such a combination might, indeed, be necessary.

Consider three programs xyProg, xProg, and yProg. Let xyProg be such that, for some function f, it maintains $x = f(y)$, and that x only occurs in assignments to x. The latter circumstance implies that x can be regarded as auxiliary variable and, hence, can be eliminated; let yProg be the result of this elimination.

Let xyProg furthermore be such that y only occurs in assignments to y or in guards. As a result, after replacing in xyProg each guard $g(y)$ by a guard $g'(x)$ not containing y, we get a program in which y can be regarded as auxiliary variable and, hence, can be eliminated. Let the result of this elimination be xProg.

Suppose further that in xyProg we can prove

(i) that each guard $g(y)$ where it occurs implies $g'(x)$

(ii) that in no alternative construct of xyProg the guards are so strong that abortion may occur

(iii) that in each repetitive construct the disjunction of the guards $g(y)$ equals that of the corresponding $g'(x)$. (Condition (iii) can be dropped when, à la Hehner, repetition is expressed by means of tail recursion: another reason to consider <u>do</u> odd!)

Under the above assumptions each computation possible under control of yProg corresponds to a computation possible under control of xProg. The transition from xProg to yProg has only reduced the degree of nondeterminacy: correctness of xProg implies correctness of yProg, partial correctness of xProg together with a proof of termination of yProg proves the latter one totally correct.

* *
*

The reason why the reduction of the nondeterminacy and the transition from x to y may have to be combined is the following. On the one hand the value of y may be needed to describe how the nondeterminacy is to be restricted : x may be a bag of values into which values can be put and from which an "arbitrary" value may be taken away, y may be a stack with the same contents, but enforcing a last-in-first-out regime. On the other hand we may be unable to introduce a y satisfying $x = f(y)$ before the restriction of the nondeterminacy, since the range of $f(y)$ may be smaller than the values of x that are permissible in xProg: one of the purposes of the restriction of the nondeterminacy may be to rule out those values of x for which the equation $x = f(y)$ cannot be solved for y . Hence the possible need for combination.

* *
*

In the naive program transformation systems referred to in EWD636, an obviously correct program should be transformed into an efficient one by applying a sequence of semantics preserving transformations, each of them to be taken from a library of such things. Such a sequence, however, is logically equivalent to a mechanical verification of the theorem that the final program is correct. In the case of efficient programs —which are really a piece of logical brinkmanship— that theorem is often deep, and hence deriving a program in such a way can be expected to be as painful as mechanical proof verification. Hence my doubts. Jonkers's transformations, however, don't come from a library and the invariance proof of $x = f(y)$ gives the opportunity of inserting a subtle argument to be provided by the inventor of the transformation. The latter way of program development by transformations is no longer naive and, therefore, much more realistic.

I hope that in the above Jonkers still recognizes his third chapter. After a mild weekend it is cold and very windy.

[0] Meertens, L.G.L.T., Abstracto 84: the next generation. Proceedings of the 1979 Annual Conference of the ACM, DETROIT (1979), 33-39

[1] Jonkers, H.B.M., Abstraction, Specification and Implementation Techniques, Mathematical Centre Amsterdam, 1982. (Thesis Technical University Eindhoven)

Plataanstraat 5
5671 AL NUENEN
The Netherlands

15 February 1982

prof. dr. Edsger W. Dijkstra
Burroughs Research Fellow