Correction to "An Approach to Register Number Determination Based on Simulation of Register Allocation via Graph Colouring"

Bojana Dalbelo Bašić

In the issue, June 1994 of CIT, several typographical errors appeared on page 114. The corrections are as follows.

We apologize to the author and the readers.

1. Concept of interference

Compiler front-end produces an intermediate code, i.e. a low-level language defined for an abstract computer (thus machine independent) which is easily transformed into the machine or assembly code. The intermediate code is very convenient for transformations and therefore it is the interproduct of optimising compilers. The primary difference between the intermediate code and the assembly code is that the intermediate code does not specify the registers. The intermediate code preferred by many compilers is the *three-address code* (Aho et al., 1986). Three-address instruction has the form: $a := x \,\mathrm{op}\, y$, where x, y can be constants, names of variables defined by the programmer, or names of compiler generated temporaries, and op being an arithmetical or logical operation. As register allocation procedure is performed on an intermediate code, intermediate code names are candidates for residing in registers and they are often called symbolic registers, names, temporaries, or variables. In this paper we shall be using the term *variable.*

The instruction $a := x \,\mathrm{op}\, y$ *defines* a and uses x, y (Aho et al., 1986). A sequence of consecutive instructions of the three-address code entered only at its beginning and whose control flow is sequential, without halt or possibility of branching, (except at the end of such a sequence), represents a *basic block* (Aho et al., 1986). A directed graph with nodes representing basic blocks and with directed edges representing control flow between blocks, makes a *flow graph.*

A simple code generator takes a sequence of three-address instructions which form a basic block and generates the target code assuming that all register values must be stored in memory when moving across basic blocks boundaries. Register allocation inside the basic block or a smaller sequential part of the code is *local register allocation* (Aho et al., 1986) and it can be solved efficiently but we are then forced to store register values at the end of each basic block. *Global register allocation* helps reduce the number of LOAD and STORE instructions by defining which variables will stay in the register across block boundaries, so registers are actually allocated for the entire procedure.

The fundamental terms related to register allocation are *variable life* and *interference of variables.* A variable is *live* at a given point in a program if it is previously defined and if there is a path in the flow graph from this point to a certain usage of this variable, otherwise variable is dead (Aho et al., 1986). If two variables

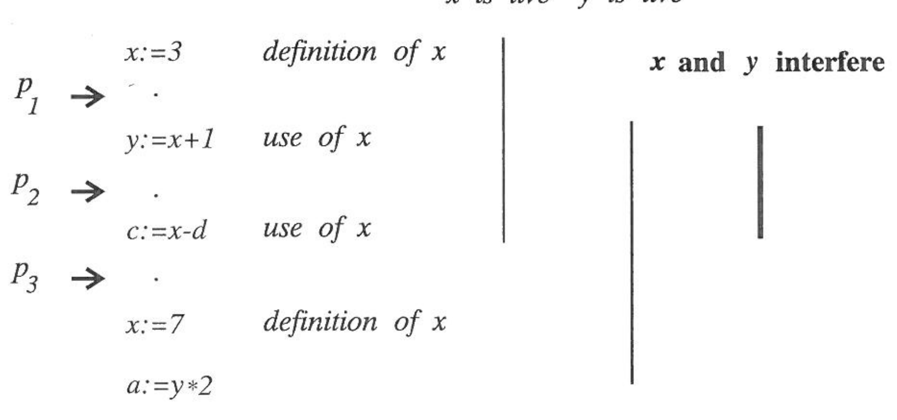

Fig. 1. Example of variable life and interference. Variable x is live at points p_1 and p_2, dead in the point p_3