SETL Newsletter # 144

J.T. Schwartz
February 25, 1975

Interprocedural Live-Dead Analysis

1. Introduction

Live-dead analysis is often carried out only intraprocedurally, in part because its principal ordinary application is to register allocation; to save a few store operations will ordinarily reduce the cost of a subroutine return only slightly, and just to secure this modest gain the extra work and complication of an interprocedural live-dead analysis might be considered excessive. In SETL however live-dead analysis plays a more important role, in that live information enters into the 'Destructive Use Condition' stated in Newsletter 131, so that the lack of accurate global live information can force large objects to be copied unnecessarily. For this reason, an interprocedural live-dead algorithm is useful in connection with SETL optimisation. Actually, we require information which is more detailed than that provided by customary live-dead algorithms; specifically, for each routine *rout*, and for each basic *block* of *rout*, we want to know the set of all local and global variables which are live at the entrance to rout. The present newsletter will describe such an algorithm.

2. Description of an Interprocedural Live-Dead Analysis Algorithm

We assume that a program P is given, and that global data-flow information has been developed for P, in the usual form of two functions *ud* and *du* which link ivariables (uses) to corresponding ovariables (definitions) and vice-versa. Using this information, and applying what is essentially transitive closure, we develop a list of all the live ovariables of P; an ovariable is live if it is used in an operation other than a simple assignment, or if it is used in a simple assignment whose output ovariable is live.

(Note that we might say, somewhat more accurately, that an ovariable is live if its value is output, or if it is linked to a use in an operation whose output is known to be live. But as a matter of fact we give all operations other than simple assignments 'benefit of doubt', since the only operations we really suspect are those which transmit arguments to sub-procedures, and we assume that these operations are represented by (nominal) assignments.)

Next, processing all the routines *rout* of P in sequence, we build up a comprehensive pair of maps, the first sending each block B of P into the set *live(B)* of all variables live on entrance to B (but with flow paths that pass through a sub-procedure 'return' instruction ignored); the second defining the set *globret(B)* of all global variables which reach a return statement along an appropriately clear path starting at the entrance to B. These two maps are developed using what may be thought of as two calls to a flow-tracing routine *reaches*, of conventional construction. (For efficiency if not for clarity, it is possible to combine these two calls into one). As input parameters, *reaches* requires functions defining the variables killed by and the variables used within each block of *rout*; and for blocks containing subprocedure calls, this information must reflect global variables used in, live (but only live) arguments used by, and global variables killed within the target routine of any such call. There is no problem about making this information available; indeed, the interprocedural data-flow tracing algorithm described in Newsletter 134 will provide it. On the second call to the routine *reaches* we transmit as parameter a dummy variable-use mapping which associates a null set of uses with every block of *rout* not ending in a 'return' statement, but associating with each such block a nominal ('post return') use of every global variable; it is easily seen that *reaches*, called in this way, will develop the desired *globret* information.

The remaining part of our work is to expand our initial estimate of the set of variables live on entrance to each particular block, in a way taking 'post-return' uses of global variables into account. (Note that variables explicitly assigned to subroutine parameters from within the subroutine, even by simple assignments, are given 'benefit of doubt', i.e., assumed to be live prior to the point of assignment, since it is assumed that explicit assignments of this kind will not be inserted unnecessarily.)

To take post-return uses into account, we invert *globret*, to get a mapping sending each global variable into the set of all blocks from which a return statement can be reached along a path clear for the variable in question. Using (essentially inverting) the map *live*, we also build up a map sending each global variable v into the set of all call instructions on return from which v is live. The set of pairs defining this map defines the initial state of a workpile; we process an item <v, block> from this workpile by adding v to *live(block)* if $v \in$ *globret(block)*, and by generating new workpile items as necessary if some predecessor of *block* ends in a call instruction. When this workpile-driven transitive closure process completes, *live* will have taken on the significance we desire.