Please feel free to correct me/delete or change any answer I have, just getting the ball rolling so others can add. (ChatGPT does not help in this module lol).

1a) ii) Unspecified Behaviour – It is not UB, but it is unsequenced based on the non-explicit call order of foo and bar in print, a compiler would have a given configuration on which order to call the function arguments in this instance (?). There is also implementation defined behaviour as the printed string will not end with a newline character, therefore whether or not the string is actually printed will differ between implementations. If a newline character is printed later, whether or not a space will be printed at the end of the line is also implementation-defined behaviour.

b) Go through the different order scenarios for foo and bar in the respective calls (I.e arg1 called then 2, etc) (?)

There are 2 different behaviors depending on the evaluation of arguments in the first ‘print’ call:

1. foo() -> bar() results in the output: 1 1 1 1
2. bar() -> foo() results in the output: 2 2 2 2

There is also an implementation-defined behaviour where nothing is printed, as the output string does not end with a new-line character.

Note that the order of evaluation of arguments in the second ‘print’ call doesn’t impact the output as it is guaranteed x != 0 after the first ‘print’ calls

c) Not necessarily. There is no formal guarantee that the configuration of sequential behaviour (I.e arg1 then 2) in one optimisation level of a compiler will be consistent across all optimisation levels. The compiler on unspecified behaviour is obliged to choose whichever order as both are technically valid. Since this is unsequenced it may not even be guaranteed to be a consistent ordering between given runs. (?).

d) GCC flagging literally anything and everything that is optimised is sure to have some correctly identified as ‘unstable’ in the warning logs. However given the large number of false positives flagged as warnings, programmers will learn to ignore these. Stack’s report logs a subset of the optimisations that GCC logs, however Stack will not be able to flag all optimised instances (for large programs) due to the approximation methods used on the well-defined prog assumption (I.e doing it over dominators from a single function), potentially missing conditions or unstable code in function calls external to it. In addition to this, STACK will miss unstable code arising from violations of strict aliasing or uninitialized variables, as well as missing consequences of unsequenced computation. This is because STACK chooses to delegate these to the compiler frontend (Which flags these anyway). Macros also ignored. Queries that result in solver timeouts will also be missed by STACK. (?).

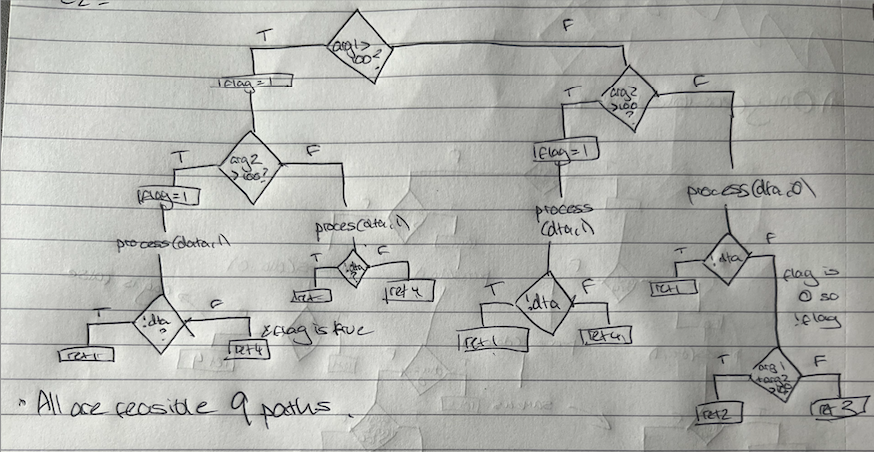
e) See above. Also compilers across different optimisation levels could differ in their sequencing choices as unspecified behaviour can go either way, across compilers.

Because Stack works on IR generated by LLVM, and IR is not ambiguous. The ambiguity for unsequenced modifications is already lost at this stage.

f) Query of UB conditions accumulated and masked one at a time, if the new query becomes satisfiable the masked UB condition is kept, and so on. (?). I thought this was about the reachability condition

2) zzzz

a) Remember that we only fork execution on symbolic branches only. Assuming no optimisations and redundant path eliminations, we have 9 feasible paths. (Someone double check this one pls ?)



b) Notice that for any path which involves setting flag to 1, arg 1 and arg 2 are not read again, they can be considered dead on these paths. Redundant Path Elimination allows us to eliminate paths if the transitive closure of the constraints/memory read are the same and aren’t different across live variables. We hit the same program point (process(data, 1)), across constraint sets that involve memory that is no longer used, therefore we can collapse these into two unique paths (I think). Process(data, 0) is of course different in that respect so incorporating the 3 paths from that we have 5 paths with the optimisation (someone double check this pls ?).

We can also collapse the true branch on !data on process(data, 0) as well since arg1 and arg2 don’t impact it, we collapse this into the other ret 1 paths, so this reduces it to 4 as stated in the comments. Same constraint sets over live variables (data == 0).

C) (All vars ({arg1, arg2, flag, data}) live? There exists paths with no redefs in the scope of process() where the vars are used at some point, idk)

D) I think this is meant to be 4

Would this make a difference? Presuming we are doing the analysis along each execution path we reach the same conclusion. (As with doing standard redundant path elimination).

E) (Unsure) - {arg1 > 100 and arg2 > 100} can be detached from {data == 0} and {data != 0} as there is no inherent dependence between data and the first two variables, constraints relating to data will not invalidate the solutions to arg1 and arg2.

F) By detaching constraints involving redundant variables, into subsets of relevant constraints that have likely been cached already, the cache is being much more optimally used and you are viable to cache hit more, reducing the load on the solver.