GEHANI80

Gehani, N.; Generic Procedures: An Implementation and an Undecidability Result; Computer Languages, Volume 5, pp. 155-161.

A generic procedure is one in which some or all information about the types of its formal parameters is unspecified, allowing one procedure to handle the work of many non-generic procedures. This is especially useful with modern languages where the user can define new data types. Generic procedures also make portability across types possible, and provide a convenient means of abstraction. Despite their advantages, generic procedures have not been popular because they typically have inefficient implementations, and allow for little compile time checking.

Gehani briefly explains a syntax that has been proposed for generic procedures in Pascal, and describes an efficient implementation method. He then points out a major problem with this scheme and suggests some possible solutions.

The notation presented is quite simple. It allows the entire type of a formal parameter to be unknown, or just some part of it (for example, the size of an array type). It also allows one to use the generic type for local variables and to reference it in boolean expressions, which enables a conditional branch dependent upon the type of a parameter. Also, a generic type or some part of it may be restricted to a certain group of types, to avoid having to write extra tests for a type with which the procedure shouldn't be called.

The suggested implementation involves compile-time conversion of a generic procedure to a procedure with specific types, once for each call to the generic procedure. Of course, calls using types for which a procedure has already been generated need not cause the creation of another version. This creation of multiple versions of a procedure may appear inefficient, but of course the user would have to write the versions otherwise.

As previously mentioned, the processing of generic procedures is done at compile time. However, there are certain recursive call sequences that would result in the creation of an infinite number of versions of the procedure. This is called infinite attribute generation. The author proves that no algorithm exists for detecting this problem, by performing a reduction on another undecidable problem.

It is concluded that there are two possible solutions to this problem. The processing of procedure calls that result in recursive calls with a different set of parameter types could be postponed until run time. This would eliminate infinite attribute generation, but involves added runtime overhead. Alternatively, we could disallow certain recursive procedure calls that might cause infinite attribute generation. This may seem very restrictive, but the author notes that the types of calls that would be thus prohibited are expected to be rarely called.