A Functional Macro Expansion System for Optimizing Code Generation: Gaining Context-Sensitivity without Losing Confluence

Eero Lassila*
Digital Systems Laboratory, Department of Computer Science and Engineering
Helsinki University of Technology
P.O. Box 1100, FIN-02015 HUT, Finland
eero.lassila@hut.fi

As the machine-specific macro assemblers offer little support for hierarchical program abstraction, they are rather weak code-generator writing tools: reusable optimized code libraries cannot be easily constructed. The underlying reason is the lack of modularity. For instance, if some assembly macro M uses some register R as a temporary data storage, the programmer has to be most cautious when calling M: the macro expander is not able to issue a warning that the value possibly already stored in R will be lost during the execution of M. Clearly, this problem cannot be solved without *context-sensitive* macros.

The traditional way to obtain some context-sensitivity is to use global expansion-time variables—in addition to the explicit macro arguments—for intermacro communication. For instance, when the contents of some memory location are loaded into the register file, a global variable may be used for recording the memory address, so that a possible subsequent load from the same memory location can be eliminated. Still, global variables do not provide modularity, and their use is subject to some basic limitations: information can be passed only from left to right and, in practice, only within straight-line code segments. Moreover, in the case of global variables we have to stick with the left-to-right and depth-first expansion; but when macro arguments are the sole communication device, the final expansion result is independent of the expansion order of the individual macro calls. Thus, global variables are incompatible with the Church-Rosser or *confluence* property.

Why should an assembly-level macro system be confluent? First, parallel implementations become possible: at any time, all the leaf nodes of the expansion tree (which originates from a single macro call) can be processed in parallel. Second, code maintainability is enhanced: if some macro definition is modified, this change propagates only to such nodes that are descendants of a call of the modified macro (with global variables, in contrast, the change could propagate to all the nodes on the right of the call). Third, it becomes straightforward to augment the macro expansion with external global optimizations: we may introduce an intermediate optimization layer inside the expansion tree, because we are free to select such an expansion order that all the macro calls above the layer are processed first.

Our idea is that each macro call is treated as a logical module rather than a text string. The module implementation, i.e. the result of the conditional expansion, depends on the expansion-time *interface* of each macro call. The interface is constituted by *domain-specific attributes*, whose expansion-time values the macro writer can examine. These values are determined by the *environment* of the macro call in a way that is invisible to the macro writer (for instance, program flow analysis may be necessary). Confluence requires that the environment of any macro call node N in an expansion tree consists of only *such nodes that are guaranteed to be present in the tree whenever N is* (that is, all the ancestors of N with all their *one-step* descendants). Moreover, the expansion has to be *functional* with no side effects: the expansion of N must not change the environment of N.

The introduction of the domain-specific attributes makes the implementation of the macro expander *domain-dependent* (in contrast to the conventional general-purpose macro processors such as the C preprocessor). As suggested above, the domain we are interested in is the synthesis of machine-level computer programs. Hence, our main domain-specific attributes are the following: Which addressing modes are used for passing run-time data into and out from the particular instance of the macro called? What is the available temporary storage (including the set of free registers) at the point of the macro call? What possibly useful knowledge does the expander already have about the run-time contents of the registers at the call point?

At the moment, our concrete contributions are a formal macro expansion model [2], which crystallizes the idea sketched above, and a simple demonstration prototype [1, 2] of the suggested retargetable macro assembler, which then applies the model to machine-level program synthesis. The prototype and some related publications are available; please see `ftp://saturn.hut.fi/pub/reflex/README`.

*Advisor: Prof. Leo Ojala (same affiliation).

References

[1] Eero Lassila. A macro expansion approach to embedded processor code generation. In *Proceedings of EUROMICRO'96 Conference*, pages 136–142. IEEE Computer Society Press, 1996.

[2] Eero Lassila. Towards optimizing code generation by domain-sensitive macro expansion. Report A42, Digital Systems Laboratory, Helsinki University of Technology, Espoo, Finland, January 1997.