Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

414

Antoni Kreczmar
Andrzej Salwicki
Marek Warpechowski

LOGLAN '88 – Report on the Programming Language

With the collaboration of Bolesław Ciesielski, Marek Lao, Andrzej Litwiniuk, Teresa Przytycka, Jolanta Warpechowska, Andrzej Szałas, Danuta Szczepańska-Wasersztrum

Foreword by Hans Langmaack

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo Hong Kong

Authors
Antoni Kreczmar
Andrzej Salwicki
Marek Warpechowski
Institute of Informatics, University of Warsaw
PKiN, room 850, P-00901 Warsaw, Poland

CR Subject Classification (1987): D.3.2

ISBN 3-540-52325-1 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-52325-1 Springer-Verlag New York Berlin Heidelberg

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210 – Printed on acid-free paper

Foreword

It is a pleasure for me to comply with the editors' desire to write a Foreword to the report on the object oriented programming language LOGLAN '88 which has been created by the informatics research group of A. Kreczmar and A. Salwicki at Warsaw University. The authors have indeed succeeded in defining and implementing a fully typed programming language of the ALGOL-PASCAL-family where more modern notions like objects, inheritance, processes and communications are harmoniously integrated with more established concepts like block structure, static scoping and higher functionality. The Warsaw group has come up with a characteristic semantical and implementational perception of the notion object which strongly supports LOGLAN's homogeneous language design.

In this foreword, I would prefer not to describe all aspects of this language and compare them with all other existing approaches; this would overcharge me at the moment since object oriented programming is an exploding area. I would prefer to be a companion of our Warsaw colleagues, to say a little about my personal acquaintance with LOGLAN and to address some of its language constructs, their semantics and implementations.

Already in the seventies, the Warsaw group started out from SIMULA 67, a language created in the sixties by O. J. Dahl, B. Myrhaug and K. Nygaard at Oslo University. SIMULA 67 already includes the idea of inheritance, but it is required that inherited classes must have the same module nesting depth as the inheriting ones. The Warsaw people call this phenomenon one level inheritance.

The Norwegians probably had several reasons for this restriction. One predominant reason is: Many level inheritance does not seem to be needed so often in applications. Two further reasons might have been, first, it is rather weary to define a reasonable and natural, i.e. ALGOL-like or static scope semantics for many level inheritance, and second, if one likes to retain Dijkstra's display register implementation technique then registers must be reloaded several times when execution takes place inside the same inheritance chain, e.g. a block, class or procedure body which has been extended by inheritance classes. For one level inheritance this reloading is not necessary; efficient implementation is simple and may follow traditional techniques as demonstrated by SIMULA 67.

But the Warsaw group felt strongly that one level inheritance is too tight a corset. For example, it forces programmers to write unnecessary copies of classes by hand, especially when programs are to be changed or corrected. One level inheritance hinders building up a flexibly usable systems library of classes. The language BETA, another successor of SIMULA 67, which has been defined by researchers at the universities of Aarhus and Oslo independently of the Warsaw group, has many level inheritance too.

So a many level prefixing semantics for LOGLAN has been defined at Warsaw and the known display register implementation technique has been drastically modified such that register reloading in prefix chains is no longer needed.

But there was a price to pay. The semantics was not fully natural, with fully static scope, and it was not invariant against bound renamings. This so-called quasi static

scope semantics behaved between static scope as in ALGOL or PASCAL and dynamic scope as in early LISP. Furthermore, the original semantics definition did not fully satisfy aesthetic requirements because of its reference to an implementing machine with a run time stack to store activation records.

These drawbacks have been eliminated in close cooperation with our Warsaw colleagues. We have defined an operational, static scope, ALGOL-like rewrite or copy rule semantics. Rewrite rules for LOGLAN '88 are to be defined not only for procedure and function calls, but also for class generations and block entries. This definition style remains fully at the programming language level without reference to any implementation.

Furthermore, we have observed that static scope semantics does not need to be inefficient. On the contrary, it is much more efficient than quasi static scope semantics with its remaining dynamic scope elements although the latter semantics is oriented at a specific implementation. Quasi static scoping needs as many display registers as there are modules in a LOGLAN program, say μ modules, whereas for static scoping the number of necessary display registers can be bounded by the maximal module nesting depth ν, which is usually much less than μ, and still no reloading inside a prefix chain is demanded.

For implementing BETA, which also adheres to the static scoping philosophy, S. Krogdahl has proposed to construct a code generation optimizer which makes display register reloading for the execution of any inheritance chain more efficient. In the light of Krogdahl's proposal our observation can be formulated in the following manner: reloading can always be optimized in such a way that at most ν registers need to be loaded when an inheritance chain is entered, and reloading is not necessary at all inside a chain. We could never do better because SIMULA and ALGOL need exactly this number ν of registers.

In his dissertation M. Krause has proved the correctness of the novel implementation technique. In addition, A. Kreczmar and M. Warpechowski have come up with a very nice and elegant axiomatic theory on static and dynamic algebras for which LOGLAN '88 programs are models. Both this theory and the new implementation technique are an outflow of considerations about what static scoping really means.

LOGLAN '88 has only mono inheritance and does not provide multiple inheritance as languages like PARAGON, SMALLTALK or ADA do. BETA does not provide multiple inheritance either for the same reasons: an acceptable and consistent semantics and a good implementation technique has still to be found out.

But critics should be fair towards LOGLAN with respect to missing multiple inheritance. Module nesting is another implicit direction of inheritance such that LOGLAN really features two dimensions of inheritance. Both theoretical investigation and existing efficient implementation demonstrate that these two combined inheritance dimensions allow a very clear and satisfying treatment. Other languages have drawbacks also: PARAGON remains with one level inheritance as far as we can see, SMALLTALK has no module nesting, and the language definition of ADA does not allow exploitation of the power of proper inheritance chains although they are available in theory.

I believe that intensive studies of static scoping in object oriented languages will eventually result in appropriate semantics definitions and implementations of languages

where multiple and many level inheritance, module nesting and static scoping occur simultaneously. Flat languages without module nesting like SMALLTALK are much better off and can avoid many semantics and implementation problems. But in my eyes, flat programming style should not be the software engineering future. Sure, in connection with module nesting, global output parameters must be treated with great care (sideeffects), but input parameters are much more often needed. And global input parameters are quite harmless and even advantageous. They save much program writing work and their parameter transmissions are more efficient than local parameter transmissions.

Since LOGLAN '88 allows functional arguments for procedures, functions and classes and functional results of functions, this language goes noticably beyond many other practically usable programming languages. LOGLAN '88 has the full power of higher functional programming languages with typing, and the several existing LOGLAN implementations demonstrate that higher functionality does not create serious implementation difficulties, not even in connection with object orientation, coroutines and processes.

Reasonable modern programming languages should allow procedures and functions as arguments. Among other advantages, such parameters and their transmissions represent a simple efficient control mechanism for stack automata activities of arbitrary complexity. We should not forget that without such parameters control has often to be done by expending data manipulation and inquiries.

A most appealing feature of LOGLAN '88 is its incorporation and treatment of processes. SIMULA 67 already has coroutines the syntactical structure and dynamical behaviour of which are somewhat close to processes. So it is reasonable that the Warsaw group has come up with a most elegant and harmonious integration of processes in LOGLAN '88. Processes are objects as classes, procedures, functions and coroutines are. Processes are generated and assigned to appropriate variables similar to classes and coroutines. The authors of LOGLAN have provided synchronous and asynchronous communications by so-called alien procedure calls and send procedure statements, respectively. LOGLAN's alien calls generalize ADA's rendezvous concept which BETA has employed also.

The object oriented programming language LOGLAN '88 which is put forward in this report has been implemented on quite a series of computers and processors, as the authors point out in their preface. LOGLAN '88 is a practically usable programming language. So the informatics community is invited to experiment with this language, especially in the areas of program structuring and communicating processes in order to get further experience, to compare with other approaches and to stimulate discussions of programming language and software engineering concepts.

Kiel, September 1989 Hans Langmaack

CONTENTS