Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel

Specification and Implementation Problems of Programming Languages Proper for Hierarchical Data Types

Bericht Nr. 8410
Dezember 1984

A. Kreczmar
A. Salwicki
Institute of Informatics
University of Warsaw
PKiN p.o.box 1210
PL-00-901 Warsaw

M. Krause
H. Langmaack
Institut für Informatik
und Praktische Mathematik
Universität Kiel
Olshausenstr. 40
D-2300 Kiel 1

Abstract

LOGLAN is a programming language proper for hierarchical data types. LOGLAN is an extension of SIMULA-67 and especially allows prefixing of modules by classes at many levels. This language construct causes semantics specification and implementation problems. In order to study these problems the programming language Mini-LOGLAN is introduced which is a smallest extension of ALGOL-like languages that allows prefixing. Based on the notion of original prefix elimination an algebraic pure static scope semantics of Mini-LOGLAN-programs is given. By means of complement modules and their unique existence a new principle of associating lists of display register numbers to modules is introduced. The number of necessary display registers is bounded by the height of the nesting tree of program modules. The proposed scheme of addressing does not cause display register reloadings while computing in one prefix chain. The designed run time system implementing pure static scoping admits a more efficient implementation of many level prefixing than the existing implementation of LOGLAN with its quasi-static scoping does.

Keywords

LOGLAN, SIMULA, class, many level prefixing, hierarchical data types, static scoping, algebraic semantics, complement modules, implementation, run time system, display registers.

Contents

0. Introduction

There are many situations in programming which need an appropriate software tool. Let us quote some cases.

1. Abstract data types. Following Hoare [Ho72] one can find his advice of factorization a convenient and useful principle. Let us recall what the principle says: Whenever possible split any reasonable "closed" piece of software into two modules: An abstract program accompanied by a module implementing the data type (i.e. representation of data and operations on them). The advantages of the factorization are easily seen. One can use the implementing module for several abstract programs and/or one can retain the abstract program and change the implementing module in order to gain better efficiency. When we think of separate compilation of modules, the principle of factorization seems to be a good advice. However, there are only a few languages supporting this style of programming.

2. Enforcing certain rules or axioms. The best example is the protocol of mutual exclusion of entry procedures of a monitor.

3. Description of families of data structures.

3.1. It is frequently so that we treat a declaration of a data type as a description of the set of objects which can potentially be constructed and memorized in a computer. In many situations there is a need to develop a hierarchy of (potential) sets of objects. E.g. in the automatization of a bank we must define a hierarchy of various types of records.

3.2. Similarly one can think of hierarchies of abstract data types. Suppose we have defined a problem oriented language as a data structure, an algebra A extended in various ways by structures B,C,... In this way one can arrive at a tree-like structure of problem oriented languages, cf. simulation class in LOGLAN.

3.3. In programming we meet often a need to define and implement dynamic systems in which objects can also play an active role (realized either as coroutines or processes).

4. Factorization of algorithms. Sometimes two or more algorithms have common initial parts (cf. insert, member, delete in binary search trees). In such situations it is natural to extract the common part in order to avoid repetitions of text. Obviously one can achieve the desired result with the help of procedures, but prefixing all the procedures by a common prefix would be also interesting.

Regarding the situations 1.-4. we see that in almost every case we can achieve the desired goal by means of prefixing.Prefixing which can also be explained as a rule of composition of modules has been invented by O.J.Dahl, B.Myhrhaug and K.Nygaard [Da70] and introduced in SIMULA-67 for the first time. In order to understand prefixing one has to be acquainted with the notion of class (again SIMULA-67 was the first language which incorporated classes).

Prefixing is a two argument operation on modules of programs. The prefix should be a class, the prefixed module can be of any kind: class, procedure, function or block. Roughly speaking the result of prefixing is the module obtained by concatenation of the declarative parts of two modules and by enclosing the statement part of the prefixed module by the prologue and epilogue coming from the prefixing module. The details will be explained later. What is more difficult to accept at a first encountering with prefixing is that the result is not a visible module. In some sense we operate in a free algebra of modules with the prefixing operation, i.e. the module

<name> : <prefix identifier> <prefixed module>

represents the result of prefixing.

This form of program construction has an unexpectedly broad spectrum of applications. In fact, we can not say that all possible advantages of prefixing are known already.

The reader should not be misled by a first impression: The concatenation rule can be explained in terms of textual operations, but the realization should not be done by textual operations.Let us recall the analogy between the copy rule for procedures and implementations of procedures in computers.

The history of prefixing can be traced back to SIMULA-67. This attractive software tool has been overlooked for years and the community of software engineers had poor conscience of the possibilities offered by prefixing.

Before we shall pass to further history let us mention a few drawbacks of SIMULA's concept of prefixing. In SIMULA there are two system classes which serve as problem oriented languages: SIMSET and SIMULATION which is prefixed by SIMSET. There is no tool for enlarging the set of system classes however. SIMULA has also a restriction: Both arguments of prefixing operation must be brothers or cousins in the tree of nesting structure of program modules (same level of prefixing), they cannot be in a nephew-uncle relation (multi-level prefixing). Due to this limitation there is no chance to extend the library of system classes. Also separate compilation of modules is difficult and of limited application due to the same reason. LOGLAN a programming language designed and implemented at the Institute of Informatics, University of Warsaw, abandons this limitation. It has turned out however that

1. it is not clear how to understand the prefixing operation if the restriction is abandoned,
2. it is difficult to find an efficient and correct implementation of prefixing by a computer system (compiler plus runtime system).

S.Krogdahl [Kr79] has discussed problems concerning many level prefixing and its implementation. There have been long studies and discussions in the Warsaw group. A first solution has been proposed in 1979 and realized in 1981 by a team led by A.Kreczmar. The results were interesting and of commercial value.

In 1983 H.Langmaack has observed that the implemented semantics of LOGLAN in certain situations does not behave according to the rule of static scoping and that this drawback can be overcome by a new principle of associating display register numbers to modules. His talk at the Zaborow Summer School on LOGLAN 1983 has caused broader interest. Now we present a version which contains contributions of several persons.

Part 1. begins with a presentation of Mini-LOGLAN. This is a smallest extension of the concepts of ALGOL- or PASCAL-like languages which admits prefixing. Its abridged form enables to concentrate on main problems of prefixing. The important notions of prefix chain and binding between applied and defining occurrences of identifiers are introduced. Based on the notion of original prefix elimination an algebraic semantics of programs with prefixing is given.

Part 2. of the paper is divided into five chapters. Chapter 2.1 introduces the important notion of complement module and the Main Lemma on unique existence of complement modules is stated and proved. In Chapter 2.2 a new principle of associating lists of display register numbers to modules is proposed which is a crucial point in efficient pure static scope compiler construction for LOGLAN. The number of necessary display registers is bounded by the height of the module tree of a program and nevertheless the proposed scheme of addressing does not cause display register reloadings while computing in one prefix chain. These features essentially improve the efficiency of generated code and run time system as compared with the existing LOGLAN compiler which implements quasi-static scoping. Pure static scoping is not only intellectually more pleasing than other semantics proposals, it admits even a more efficient implementation. Chapters 2.3 and 2.4 contain the design of run time storage and generated code; run time system subroutines are described in Appendix G. Chapter 2.5 shows different run time systems which need less storage place.

1. Semantics specification

1.1 A contextfree - like grammar for Mini-LOGLAN

In order to enable a proper treating of semantics specification and implementation of programming languages with prefixing we should like to present in Appendix A the language Mini-LOGLAN which is a smallest extension of ALGOL- or PASCAL-like languages which allows prefixing. The grammar for Mini-LOGLAN is not complete but contains all relevant parts to talk about prefixing.

Modules are blocks, procedures and classes and they can be prefixed by classes (which have no local formal parameters in Mini-LOGLAN). Procedures need not necessarily be prefixed; their prefixing can be simulated by prefixing of their bodies written as blocks. A class initialization

$$\underline{\text{new}}\ \xi$$

can be simulated by a simple prefixed block

$$\eta : \xi\ \underline{\text{block}}\ \underline{\text{begin}}\ \underline{\text{end}}\ \eta$$

with an empty declaration and statement list. The main part of the statement list Σ of a class body contains exactly one simple control statement inner with its prologue Σ_1 and epilogue Σ_2:

$$\Sigma = \Sigma_1\ \underline{\text{inner}}\ \Sigma_2 .$$

The main part of a program piece is that part outside all inner procedure or class modules.

The grammar shows indications where non-standard, non-system identifiers occur in a defining manner. Further defining identifier occurrences are variable identifiers in variable declarations and formal parameter identifiers in formal parameter lists. All other non-standard, non-system identifier occurrences are called applied.

1.2 Basic Definitions

A syntactical Mini-LOGLAN-program (program for short) π is a finite string of lexical entities generated by the Mini-LOGLAN-grammar with <program> as its axiom. Lexical entities are identifiers (including system identifiers = word delimiters = word symbols and standard identifiers),numbers, strings and delimiters. Every program π has a length $|\pi|>0$. s is called a substring of π iff $\pi = xsy$. The couple (i,s) is called an occurrence of a substring in π (or shorter: a substring in π) iff $\pi = xsy$ and $i = |x|+1$. We write also ^{i}s and even s for (i,s) as soon as no misunderstandings are possible. We shall mainly talk about structured substrings in π; these are substrings in π generated by the (unique because the grammar is unambigous) structure tree of π. The generating subtrees of structured substrings in the structure tree of π are uniquely determined. Two structured substrings are either disjoint or contained in each other. An occurrence i in a program π is an integer with $1 \le i \le |\pi|$. We are especially interested in identifier occurrences $^{i}\xi$ in π (or identifiers in π) where ξ is a non-system, non-standard identifier and $^{i}\xi$ is a substring in π. Unless especially mentioned we shall understand the word "identifier" in this restricted sense.

Positions of defining occurrences of identifiers in a program have been indicated in the grammar, all other occurrences are applied ones. A module occurrence ^{i}M in a program π starts with block, class or proc and finishes with a matching end-symbol; we speak about block, class or procedure modules respectively. Modules ^{i}M in π form a tree; they have nesting levels $\nu_{^{i}M} \geq 1$; the largest module in a program is a block module $^{j_1}M_1$ and has level 1. If an occurrence i or a structured substring ^{i}s is in $^{j_1}M_1$, $j_1 \le i < j_1+|M_1|$, then it has an environment module env(i) resp. env(^{i}s), namely the smallest module ^{j}M in which i occurs, $j \le i < j+|M|$. The associated local identifier list locidl(^{j}M) of a module in π is the ordered list of all defining identifier occurrences $^{i}\xi$ with env($^{i}\xi$)=^{j}M. Ordering in this list means that $^{i_1}\xi_1$ is left of $^{i_2}\xi_2$ iff $i_1 < i_2$.

1.3 Binding Function and Prefix Chains

The binding function bdfct(i,ξ) of an identifier ξ with respect to occurrence i and the binding function bdfctpref(i,ξ,$^{l}\eta$) of an identifier ξ with respect to occurrence i in a prefix chain starting with class identifier occurrence $^{1}\eta$ in π are mutually recursively defined:

$\mathrm{bdfct}(i,\xi) =_{Df}$
    if i is outside the largest module M_1 in π
    then if π is a block named by an identifier equal to the argument ξ
        then $^{1}\xi$
        else undefined **fi**
    else if ξ occurs in locidl(env(i))
        then the rightmost entry $^{j}\xi$ in locidl(env(i))
        else if $\mathrm{env}(i)={}^{j}M$ has a prefix identifier $^{j-1}\eta$
            then $\mathrm{bdfctpref}(j-1,\xi,{}^{j-1}\eta)$
            else $\mathrm{bdfct}(j-1,\xi)$ **fi fi fi**

$\mathrm{bdfctpref}(i,\xi,{}^{1}\eta)=_{Df}$
    if $\mathrm{bdfct}(1,\eta)$ is a defining class identifier occurrence $^{k}\eta$ with its class module $^{m}M, m=k+3$ or $m=k+2$ depending on whether ^{m}M is prefixed or not
    then if ξ occurs in $\mathrm{locidl}({}^{m}M)$
        then the rightmost entry $^{j}\xi$ in $\mathrm{locidl}({}^{m}M)$
        else if ^{m}M has a prefix identifier $^{m-1}\zeta$
            then $\mathrm{bdfctpref}\ (i,\xi,{}^{m-1}\zeta)$
            else $\mathrm{bdfct}(i,\xi)$ **fi fi**
    else undefined **fi**

If $\mathrm{bdfct}(i,\xi)$ is $^{j}\xi$ and if i and j are inside the largest module M_1 in π then the level of $\mathrm{env}({}^{j}\xi)$ is obviously $\leq$ the level of env(i).

The <u>binding function</u> of an identifier occurrence $^{i}\xi$ in π is
    $\mathrm{bdfct}({}^{i}\xi)=_{Df}\mathrm{bdfct}(i,\xi)$
and the <u>prefix module</u> of a module ^{i}M in π is
$\mathrm{pref}({}^{i}M)=_{Df}$
    if ^{i}M has a prefix identifier $^{i-1}\eta$
    then if $\mathrm{bdfct}(i-1,\eta)$ is a defining class identifier occurrence $^{k}\eta$ with its class module $^{m}M, m=k+3$ or $m=k+2$
        then ^{m}M
        else undefined **fi**
    else undefined **fi**

For pref(M)=M' we write also M ---→ M'.

The prefix chain of ^{i}M is the sequence

$$\ldots \; pref^{2}(^{i}M) \leftarrow\!-\!- \; pref^{1}(^{i}M) \leftarrow\!-\!- \; pref^{0}(^{i}M)$$

which is of length l>0 iff $pref^{l-1}(^{i}M)$ is defined but $pref^{l}(^{i}M)$ is not defined. The module levels from right to left are not increasing, obviously.

We denote the smallest strict environmental module of a module M by strenv (M). If M is named by an identifier $^{i}\xi$ in front of M then

$$strenv(M) = env(^{i}\xi)$$

or both are undefined (in case $M = M_1$).

Let M be a module occurrence of level ν_M and let

$$M_1 \longleftarrow M_2 \longleftarrow \;\ldots\; \longleftarrow M_{\nu_M - 1} \longleftarrow M_{\nu_M} = M$$

with

$$M_{i-1} = strenv(M_i)$$

be the chain of environmental (surrounding) modules of M. The total prefix chain of M is the list

$$pref^{0}(M_1)\, pref^{l_2 - 1}(M_2) \ldots pref^{0}(M_2)$$
$$\vdots$$
$$pref^{l_{\nu_M} - 1}(M_{\nu_M}) \ldots pref^{0}(M_{\nu_M}).$$

M_1 has no prefix module because a possible prefix identifier would not be bound to a defining class identifier occurrence. If we replace every module M' in the total prefix chain by its local identifier list locidl(M') and prefix the resulting list by the possible defining identifier occurrence $^{1}\xi$ in front of the whole program π then we get the so called total identifier list totidl(M).

Let us assume that all prefix chains in π are finite. Then we may characterize the binding function bdfct(i,ξ) with the help of the notion total identifier list: Let i be an occurrence in the largest module $^{i_1}M_1$ in π with $i_1 \le i < i_1 + |M_1|$. Then bdfct(i,ξ) is defined to be $^{j}\eta$ if and only if there is a rightmost identifier entry $^{h}\zeta$ in totidl(env(i)) with $\zeta = \eta = \xi$ and $h = j$.

i, ξ and bdfct(i,ξ)=${}^{j}\xi$ determine especially a minimal module minmod(i,ξ) with

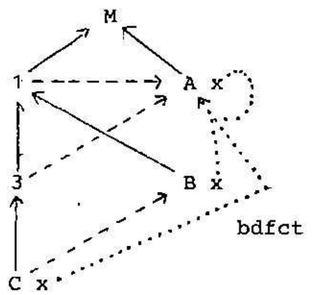
$$\begin{array}{c} \text{minmod}(i,\xi) \\ *\uparrow \\ \text{env}(i) \end{array}$$

such that ${}^{j}\xi$ occurs in the local identifier list of a module env(${}^{j}\xi$) in the prefix chain of minmod(i,ξ):

$$\begin{array}{ccc} \text{minmod}(i,\xi) & \overset{*}{\dashrightarrow} & \text{env}({}^{j}\xi) \\ *\uparrow & & \\ \text{env}(i) & & \end{array}$$

A closed program π is one where every applied identifier ${}^{i}\xi$ in π is bound to its associated defining occurrence ${}^{j}\xi$=bdfct(${}^{i}\xi$) and where all prefix chains of modules in π are finite. If all applications in a closed program "make sense" then we talk about a compilable or proper program. A closed program π may be named by a block identifier ${}^{1}\xi$ but π has no prefix.
A program π is called to be distinguished iff different defining occurrences of identifiers ${}^{i}\xi \neq {}^{j}\eta$ in π are denoted by different identifiers $\xi \neq \eta$ and free identifiers are different from bound ones; the latter condition is always satisfied in a closed program. Bound renamings of identifiers in π can always generate a so called congruent distinguished program π_d.

Appendix B shows a program example π_1 with the following environmental and prefix structure:



There are defining occurrences of x in the local identifier lists of module M and A. There are applied occurrences of x in the main parts of module A, B and C as shown in the diagram. The binding functions of these three occurrences all point into module A, no one into M, because A is the prefix of module 1 and 3. Especially, minmod of x in A is class A, of x in B is block 1, and of x in C is block 3.

1.4 Original Prefix Elimination

We shall base the semantics of programs with prefixing on the idea of prefix elimination which makes prefix chains shorter. We call this process original prefix elimination because we shall later discuss a different elimination method.

Let π be a closed program. Let in π a class declaration

(1) $\eta:\xi$ **class** Δ **begin** Σ **end** η

or a block

(2) $\eta:\xi$ **block** Δ **begin** Σ **end** η

be given, prefixed by ξ which identifies a class

(3) $\xi:\xi'$ **class** Δ' **begin** Σ_1' **inner** Σ_2' **end** ξ.

We have assumed that this class is again prefixed by ξ' what must not necessarily be the case.

Prefix elimination replaces the class (1) or block (2) by a class (1') or block (2') in the following way:

(1') $\eta:\xi'$ **class** Δ' Δ **begin** Σ_1' Σ Σ_2' **end** η

or

(2') η: ξ' **block** Δ' Δ **begin** Σ_1' Σ Σ_2' **end** η .

If ξ' is not existent in (3) then ξ' is simply not existent in (1') and (2').

Elimination of prefixes of procedures is done in an analogous way. We see especially that replaced modules remain modules of the same kind, namely classes, blocks or procedures.

Lemma 1: If π is a closed program then this prefix elimination yields a new closed program π'

$$\pi \underset{\text{pref elim}}{\vdash\!\!\!-\!\!\!-\!\!\!-\!\!\!-} \pi'$$

if a) π is distinguished or

b) the prefixed class, block or procedure η is outside any prefixed class, block or procedure or

c) class ξ has no prefix.

If π is a proper program then the prefix elimination yields a new proper program π' if

a) π is distinguished.

1.5 Discussion of Original Prefix Elimination

As an illustration of Lemma 1 Appendix C shows an example of a program π_2 which is

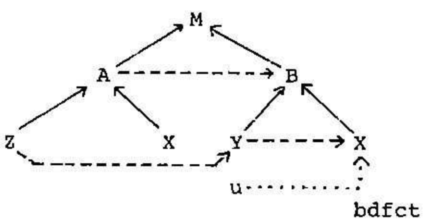
$\bar{a}$) not distinguished and where

$\bar{b}$) the prefixed block η=Z is inside a prefixed block A and where

$\bar{c}$) class ξ=Y has a prefix X.

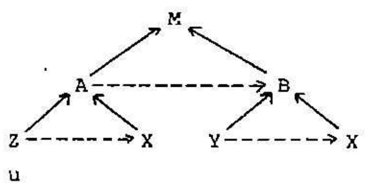
π_2 is proper, especially closed

π_2:



but prefix Y elimination leads to a program π_2' which is not proper,

π_2':

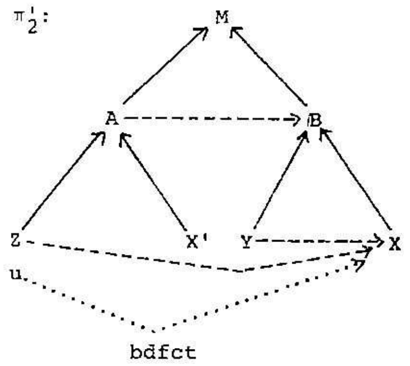


bdfct undefined

because the applied identifier occurrence u in block Z has no associated defining occurrence u.

The example from Appendix C shows that the elimination of prefices leading to ALGOL-like programs must be done with due care.

If we rename class X in block A in π_2 into class X' then after prefix Y elimination u in block Z in π_2' is bound to var u in class X in class B what is reasonable:



How influential bound renamings for the prefix elimination process are this is demonstrated by Appendix D. We apply prefix elimination to example π_1 of Appendix B and successfully eliminate the prefixes A, B and C. π_1''' has no prefixes and yields an output

(1) 2.0, 4.0, 4.0

If we would have made π_1 distinguished by a bound renaming then π_1''' would yield an output

(2) 2.0, 2.0, 2.0

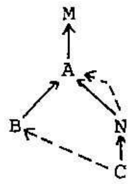
If we make not only the starting program π_1 but also the intermediate results π_1' and π_1'' distinguished then π_1''' delivers

(3) 2.0, 2.0, 3.0

The last proceeding (3) follows the ALGOL-like or pure static scope idea whereas "no renaming" (1) is often referred to as dynamic scoping. (2) represents the rationale for the first implementation of LOGLAN [Lo83] which in case of ALGOL- or SIMULA 67-like programs as π_2 [Na63 , Da70] works with static scoping and in case of programs with many level prefixing like π_1 adds on "some kind" of dynamic scoping. We call this scoping quasi-static. The new implementation of LOGLAN will follow the pure static scope strategy which is not only intellectually more pleasing but offers even a more efficient implementation as we shall see later in Chapters 2.3 and 2.4.

The original prefix elimination process cannot eliminate all prefixes. Program π_3 in Appendix E is an example of so called <u>recursive prefixing</u>, i.e. π_3 has an infinite formal execution lattice E_{π_3}, see next chapter, although there are no procedures declared in π_3.

π_3:



Bound renamings of identifiers have no influence on the identifier binding and the elimination process applied to π_3.

Recursive prefixing is a phenomenon not possible in SIMULA 67-like programs because any module and its prefixes are required to have the same nesting level there.

<u>Resumée</u> of this discussion. Original prefix elimination should only be applied to a program π if prior to every elimination step the programs have been made distinguished by bound renamings (a weaker notion of distinguity would also work, but we should not like to formulate this here).

1.6 Algebraic Semantics of Programs with Prefixing

We should like to define the semantics of Mini-LOGLAN-programs in an algebraic style [Gu81]. We view this style as an abstraction from the operational style as presented e.g. in the ALGOL 60-report [Na63]. There are no difficulties to apply the algebraic method to ALGOL-like languages even with a full procedure and function concept [La73], and now we extend this method to programs with prefixing.

Let a proper program π be given. First we form the associated formal execution lattice E_π:

We define a generating relation

$$\pi' \longmapsto \pi''$$

between certain proper programs π' and π'' . Let a distinguished program π' be given. Then we generate a program π'' by looking for one of the following three types of statements in π':

- A correct procedure statement

  call $\varphi(a_1,\ldots,a_n)$

  in the main part of π' (main program of π') outside all prefixed blocks.

  Here we apply the copy rule known already from the ALGOL 60-report. Correctness guarantees that the generated program π'' is also proper [1]. If procedure φ is prefixed by ξ then the copy rule produces a so called generated block in π'' also prefixed by ξ.

- A class initialization statement

  new η

  in the main program of π' outside all prefixed blocks.

  Here the copy rule for classes is applied which acts in an analogous way as the copy rule for procedures where the control statement **inner** in the main part of class η is to be replaced by the empty statement. If class η is prefixed by ξ then the copy rule produces a generated block also prefixed by ξ , and the gene-

---

[1] In a theory of copy rule application it is advantageous not to demand that in a proper program all procedure statements are correct; they have to be only "partially correct".

rated program π'' is proper.

- A maximal prefixed block

ξ:n <u>block</u> Δ <u>begin</u> Σ <u>end</u> ξ

in the main program of π'.

Here we apply original prefix elimination and produce a generated block ξ in the newly generated program π'' which is proper due to Lemma 1.

Now we consider the equivalence classes $[\pi]$ of congruent (boundly renamed) proper programs and define the <u>extended generating relation</u>

$$[\tilde{\pi}'] \longmapsto [\tilde{\pi}'']$$

between classes iff there are representatives

$$\pi' \in [\tilde{\pi}'] \text{ and } \pi'' \in [\tilde{\pi}'']$$

with

$$\pi' \longmapsto \pi''$$

what implies that π' must be distinguished.

The <u>formal execution lattice</u> E_π of a proper program π is defined to be

$$E_\pi =_{Df} \{[\pi'] \mid [\pi] \overset{*}{\longmapsto} [\pi']\},$$

the set of all equivalence classes generated by $[\pi]$ and $\overset{*}{\longmapsto}$.

<u>Lemma 2</u>: $(E_\pi, \overset{*}{\longmapsto})$ is a distributive lattice with $[\pi]$ as its least element.

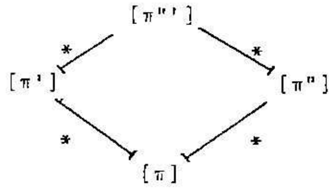
A distributive lattice is always isomorphic with a sublattice of the power set lattice $(\mathfrak{P}\, T, \subseteq)$ of a set T. In our special situation T can be chosen to be the set T_π of all those equivalence classes $[\pi'] \in E_\pi$ where the generated blocks occurring in π' are all nested in each other.

<u>Lemma 3</u>: $(T_\pi, \overset{*}{\longmapsto})$ is a tree $\subseteq (E_\pi, \overset{*}{\longmapsto})$ and $(E_\pi, \overset{*}{\longmapsto})$ is isomorphic with the sublattice $(\mathfrak{I}_\pi, \subseteq) \subseteq (\mathfrak{P}\, T_\pi, \subseteq)$ of all <u>finite initial trees</u> $I \subseteq T_\pi$. If class $[\pi'] \in E_\pi$ and initial tree $I' \in \mathfrak{I}_\pi$ are associated due to this isomorphism then the tree of all generated

blocks occurring in π'[1] is isomorphic with I'. E_π is finite if and only if T_π is finite. T_π is the so called formal execution tree of π.

Now we reduce all programs π' in E_π: We erase all procedure and class declarations and replace all remaining prefixed blocks and all remaining procedure and class initialization statements by the error statement error. Congruent programs remain congruent by this process.

The semantics $\Sigma_{\underline{error}}$ is the totally undefined state transformation and the semantics $\Sigma_{\pi'_{red}}$ of a reduced program π'_{red} is a well definable state transformation because π'_{red} is a block structured program without any procedures or classes. The state transformations $\Sigma_{\pi'_{red}}$ of all programs π' in E_π are continuations of each other simply because two different classes $[\pi']$ and $[\pi'']$ in E_π have a common supremum $[\pi''']$



in E_π. So the union

$$\bigcup_{[\pi']\in E_\pi} \Sigma_{\pi'_{red}}$$

is a well defined state transformation. This one we define to be the semantics Σ_π of the proper program π. All programs π' in E_π are semantically equivalent

$$\Sigma_{\pi'} = \Sigma_\pi$$

again simply because two classes in E_π have a common supremum in E_π.

---

[1] For technical reasons it is advantageous to call the largest block of any program π' in E_π also a generated block. So program π in E_π where no generating step has been applied has exactly one occurring generated block and this one element tree is isomorphic with the one element initial tree $\{[\pi]\} \subseteq T_\pi$.

1.7 Prefix Elimination by Transformation into Procedures

As long as we deal only with SIMULA 67-like programs (within Mini-LOGLAN) with prefixing on the same level only then successive original prefix elimination leads to programs which have no longer any class initialization, prefixed block or prefixed procedure. So all remaining classes and their prefixes have become redundant and the resulting programs may be called ALGOL 60-like ones the semantics of which is well known. This proceeding offers another way to define the semantics of programs with prefixing, but it does not work for all Mini-LOGLAN-programs because in Appendix E we have seen an example π_3 with recursive prefixing.

But there is a different prefix elimination process by transforming classes and prefixed blocks into procedures. Let π be a proper distinguished program. We are allowed to assume that there are no class initialization statements nor prefixed procedures in π.

Let a non-prefixed class declaration

$$\eta : \underline{\text{class}}\ \Delta\ \underline{\text{begin}}\ \Sigma_1\ \underline{\text{inner}}\ \Sigma_2\ \underline{\text{end}}\ \eta$$

in π be given which defines a module ${}^{j}M$ in π. Let

$${}^{i_1}\xi_1 \ldots {}^{i_n}\xi_n$$

be the local identifier list locidl(${}^{j}M$). inner indicates the only inner-statement in the main part of statement list Σ. Then the module above is transformed to

$$\eta : \underline{\text{proc}}(\eta_f);\ \Delta\ \underline{\text{begin}}\ \Sigma_1\ \underline{\text{call}}\ \eta_f(\xi_1,\ldots,\xi_n)\ \Sigma_2\ \underline{\text{end}}\ \eta$$

where η_f is a new formal procedure identifier with an appropriate specification (which we have deleted). The specification of η_f is induced by the declarations of $\xi_1,\ldots,\xi_n$ in a well known way.

Now we consider a prefixed class declaration

$$\eta : \xi\ \underline{\text{class}}\ \Delta\ \underline{\text{begin}}\ \Sigma_1\ \underline{\text{inner}}\ \Sigma_2\ \underline{\text{end}}\ \eta$$

in π which defines a module ${}^{j}M$ in π with its finite prefix chain

$$pref^{l-1}({}^{j}M) \Leftarrow\!\!-\!- \ldots \Leftarrow\!\!-\!- pref^{0}({}^{j}M), l>1,$$

and its local identifier lists

$$^{i_1}\zeta_1 \ldots {}^{i_n}\xi_n = \text{locidl}(\text{pref}^{0}({}^{j}M))$$

$$^{j_1}\zeta_1 \ldots {}^{j_m}\zeta_m = \text{locidl}(\text{pref}^{l-1}({}^{j}M)) \ldots \text{locidl}(\text{pref}^{1}({}^{j}M)).$$

Then the above module is tranformed to

```
η : proc(η_f);
    η_g : proc(ζ_1,...,ζ_m);
          Δ
    begin
          Σ_1 call η_f(ζ_1,...,ζ_m,ξ_1,...,ξ_n) Σ_2
    end η_g;
begin
    call ξ(η_g)
end η
```

where η_f and η_g are new procedure identifiers with appropriate specifications (which we have deleted). The specifications of η_f and $\zeta_1, \ldots, \zeta_m$ are induced by the declarations of $\zeta_1, \ldots, \zeta_m$ and $\xi_1, \ldots, \xi_n$.

A prefixed block is treated similarly: Let

η : ξ block Δ begin Σ end η

be such a block in π which defines a module ^{j}M in π. This block is transformed to

```
η : block
    η_g : proc(ζ_1,...,ζ_m);
          Δ
    begin
          Σ

    end η_g;
begin
   call ξ(η_g)
end η.
```

The symbols have the same meanings as before.

Now we should like to sketch a proof why the given Mini-LOGLAN-program π and its transformed ALGOL-like program are semantically equivalent in the sense of the preceding chapter 1.6 .

Let us look at a non-prefixed class

(1) ξ : <u>class</u> Δ' <u>begin</u> Σ_1' <u>inner</u> Σ_2' <u>end</u> ξ

which is a prefix of a block

(2) η : ξ <u>block</u> Δ <u>begin</u> Σ <u>end</u> η.

Let

$$i_1 \zeta_1 \ldots i_n \zeta_n$$

be the local identifier list of class ξ. The translated class and block look as follows

```
(3)  ξ: proc(ξ_f);
          Δ'
     begin
          Σ'_1 call ξ_f(ζ_1,...,ζ_n) Σ'_2
     end ξ

(4)  η : block
          η_g : proc(ζ_1,...,ζ_n);
             Δ
          begin
             Σ
          end η_g;
     begin
          call ξ(η_g)
     end η.
```

Now we compare original prefix elimination in (1), (2) and copy rule applications in (3), (4). Prefix elimination gives

```
     η : block
          Δ'
          Δ
     begin
          Σ'_1 Σ Σ'_2
     end η
```

and copy rule applications give

first step:

```
η : block
  :
begin
  block
    Δ'
  begin
    Σ'₁ call η_g(ζ₁,...,ζ_n) Σ'₂
  end
end η
```

second step:

```
η : block
  :
begin
  (**) : block
    Δ'
  begin
    Σ'₁
      (*) : block
        Δ
      begin
        Σ
      end (*)
    Σ'₂
  end (**)
end η.
```

If we assume distinguity for π and separation for class ξ and block η then both copy rule applications do not cause scope binding errors. If we assume distinguity for π and if block η is contained in class ξ then both copy rule applications might cause binding errors in the sense of static scoping. If we do not want them then global parameters of block (*) may not point into Δ' of block (**). Appropriate renamings must be done. Procedure η_g is redundant finally.

Resumée: In case of distinguity of π and separation of class ξ and block η both processes, prefix elimination and transforming rule plus copy rule applications, lead to essentially equivalent programs. All other cases

- a block η prefixed by a prefixed class ξ
- a class η prefixed by a non-prefixed class ξ
- a class η prefixed by a prefixed class ξ

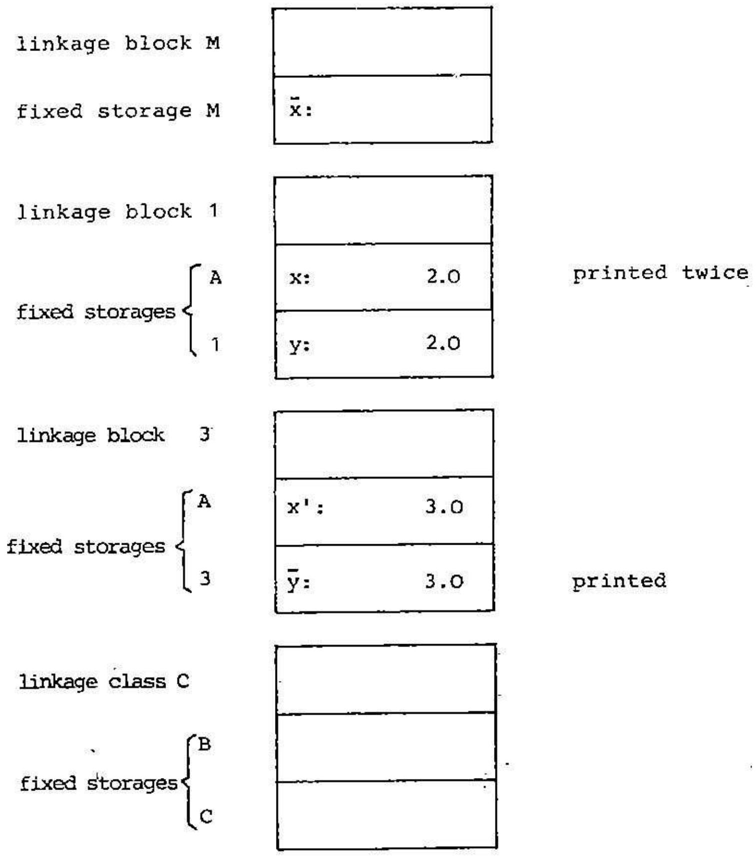
lead to analogous results. We say "essential equivalence": We have full equivalence if the control statement inner in (1) does not occur inside a loop in Σ_1' inner Σ_2'. In order to cope also with this situation the transformation process has to be a little more complicated.

Theorem 1: A Mini-LOGLAN-program and its effectively transformed ALGOL-like program (all classes and prefixes eliminated) are semantically equivalent.

Appendix F shows the transformed programs π_1 and π_3 of Appendix B and E.

2. Implementation

Designing an efficient implementation for LOGLAN with many level prefixing and pure static scope semantics is a severe problem, much severer than for ALGOL 60 or SIMULA 67. The starting idea is Dijkstra's[Di60], namely to enter activation records in a run time stack when modules are activated, to cancel them when modules are terminated and to use compile time determinable display (index) registers and offsets (relative addresses) for fast access to contents of non-formal variables. Like for SIMULA 67 incarnation (instantiations) of modules in one prefix chain shall be grouped as a so called object into one activation record and no display register reloading shall be needed when computing in and running through the main parts of modules in a prefix chain . As an illustration look at the run time stack content (pure static scoping) of program π_1 in Appendix B with its environmental and prefix structure in chapter 1.3 just before class C is terminated:



In SIMULA 67 as in ALGOL 60 or PASCAL it suffices to associate any module M of level ν_m with a list of display registers of numbers $1,2,\ldots,\nu_M$ and to associate any applied occurrence of a variable ξ with a display register numbered by the level $\nu_{env(bdfct(\xi))}$.

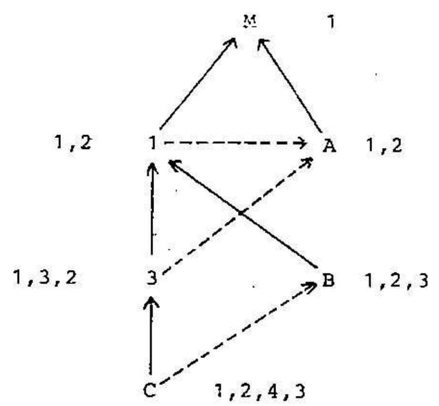
But this proceeding does no longer work for many level prefixing. Krogdahl [Kr79] discusses this for pure static scoping; he recommends in general to make reloadings of display registers when running through a prefix chain and to look for optimizations at compile time which will often be applicable.

The first implementation of LOGLAN uses a 1-1-association of modules and display registers what ends up with a total of 6 registers for π_1. But the implemented semantics is not pure static scope only quasi-static scope, see Chapter 1.5 and Appendix D.

We demonstrate in this paper that pure static scope semantics can get along with a number of display registers bounded by the maximum module level in a program such that no reloadings inside a prefix chain are necessary. So pure static scope semantics is not only the most pleasing one, it admits even a more efficient implementation than other proposals do.

Before we present more formally the general solution let us look how it works for the example from Appendix B.

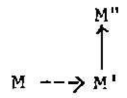
For π_1 we shall have the following lists of display register numbers:



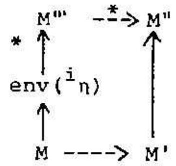
Please remark that these lists are not monotonous. Further remark that the contents of the applied occurrence of variable x in class B (x is defined in class A) are accessed with the help of display register 2 because minmod (x in B) is block 1 with level 2 which points to register 2 in the list for B and of variable x in class C are accessed with the help of display register 4 because minmod(x in C) is block 3 with level 3 which points to register 4 in the list for C. We see: The machine instructions for different applied occurrences of the same variable x in one prefix chain C--→ B may show up different index register modifications. But pay attention: These compile time phenomena do not lead to run time inefficiencies. Many level prefixing is as efficiently implementable as SIMULA 67 with its same level prefixing.

2.1 Complement modules

Let the following diagram in a proper program π be given –

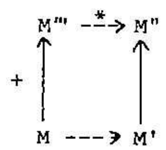


where M'=pref(M) and M"=strenv (M').
Then M' is named by an identifier $^{j}\eta$ (defining occurrence) and M is prefixed by $^{i}\eta$ (applied occurrence). Especially $^{i}\eta$ is outside M and env($^{j}\eta$)= M". So we have the minimal module M‴ =minmod(i,η) with



M‴ is the smallest module fulfilling the diagram

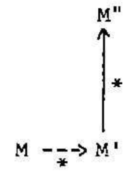
(**)



and we call M‴ the <u>complement module</u> compl(M,M',M").

Now let a diagram

(*)



be given. What is the complement module comp(M,M',M") in this case?

The diagram is a $\overset{--\to}{\longrightarrow}$ -chain between M and M" and we consider the family of all such chains. This family is finite because π is a proper program and we have no repetitions of modules in such chains. If we define that $\longrightarrow$ <u>precedes</u> $\dashrightarrow$ then we have a lexicographical total ordering of <u>precedence</u> among all $\overset{--\to}{\longrightarrow}$ -chains.

Now let us <u>replace</u> any subchain

$$\tilde{M} \dashrightarrow \tilde{M}' \longrightarrow \tilde{M}''$$

by

$$\tilde{M} \xrightarrow{+} \underbrace{\mathrm{compl}(\tilde{M},\tilde{M}',\tilde{M}'')}_{\tilde{M}'''} \overset{*}{\dashrightarrow} \tilde{M}''$$

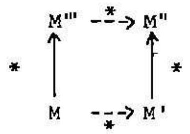
which leads to a strictly preceding $\overset{--\to}{\longrightarrow}$ -chain between M and M". Finiteness of the family guarantees

<u>Main Lemma:</u> Successive replacing ends up with a uniquely determined least preceding chain which is of the form

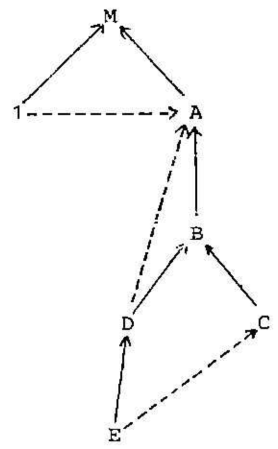
$$M \xrightarrow{*} M''' \overset{*}{\dashrightarrow} M'' .$$

M''' is called the <u>complement module</u> compl(M,M',M"). Especially compl(M,M,M")=M", compl(M,M",M")=M and if M'$\neq$M" then compl(M,M',M")$\neq$M.

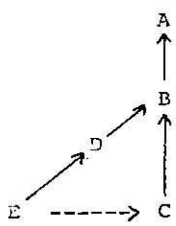
Intuitively we may say we have <u>paved</u> diagram (*) with pavestones of the type (**) and have ended up with a paved diagram



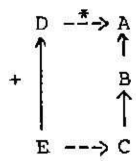
In general M''' is not the smallest module fulfilling this diagram when M, M' and M" are given. Appendix H shows a program example π_4 with an environmental and prefix structure:



The complement of E, C, A is A:



what can be found out by paving, whereas D is the minimal environment fulfilling



when E, C and A are given.

Let module M' be in the prefix chain of module M

$$M \overset{*}{-\,-\!\!\rightarrow} M'.$$

We may consider the environmental chain

$$M'_1 \leftarrow M'_2 \leftarrow \dots \leftarrow M'_{\nu_{M'}-1} \leftarrow M'_{\nu_{M'}} = M'$$

with its level list

$$1 \;,\; 2 \;,\; \dots \;,\; \nu_{M'}-1,\; \nu_{M'}$$

where M'_1 is the largest module in π: The so called *complement environmental chain* of M, M' is defined by

$$M'_1 = M_1 \overset{+}{\longleftarrow} M_2 \overset{+}{\longleftarrow} \dots \overset{+}{\longleftarrow} M_{\nu_{M'}-1} \overset{+}{\longleftarrow} M_{\nu_{M'}} = M$$

with

$$\begin{array}{ccccc}
 & M_1 & = & M'_1 \\
 & +\uparrow & & \uparrow \\
 & M_2 & \overset{*}{- - - \rightarrow} & M'_2 \\
 & +\uparrow & & \uparrow \\
 & \vdots & & \vdots \\
 & +\uparrow & & \uparrow \\
 & M_{\nu_{M'}-1} & \overset{*}{- - \rightarrow} & M'_{\nu_{M'}-1} \\
 & +\uparrow & & \uparrow \\
M = & M_{\nu_{M'}} & \overset{*}{- - \rightarrow} & M'_{\nu_{M'}}
\end{array}$$

where the single diagrams are complemented diagrams. The level list of the complement environmental chain

$$1 = \nu_{M_1},\ \nu_{M_2},\ \nu_{M_{\nu_{M'}-1}},\ \nu_{M_{\nu_{M'}}} = \nu_M$$

is strictly monotonous and is called the *complement level list* of M, M'.

2.2 Association of lists of display register numbers to modules

Every module M of level $\nu_M \geq 1$ has to be associated with ν_M distinct display register numbers

$$d_M(1), \ldots, d_M(\nu_M)$$

with $1 \leq d_M(j) \leq \nu_M$ for $j=1,\ldots,\nu_M$. So d_M is a permutation of the numbers $1,\ldots,\nu_M$. This association shall fulfill the following

Condition: Let M' be in the prefix chain of M

$$M \overset{*}{\dashrightarrow} M'.$$

Then the display register numbers $d_{M'}(j)$, $j=1,\ldots,\nu_{M'}$ are to be the same as the display register numbers $d_M(\nu_{M_j})$, $j=1,\ldots,\nu_{M'}$, of the complement level list of M, M'. Especially $d_{M'}(\nu_{M'}) = d_M(\nu_{M_{\nu_{M'}}}) = d_M(\nu_M)$.

Is such an association d_M of lists of display register numbers to modules M possible? We define d_M by induction over the lexicographical total ordering of couples (ν_M, l_M) of level ν_M and prefix chain length l_M.

Induction beginning $(\nu_M, l_M) = (1,1)$:

$$d_M(1) =_{Df} 1$$

is the only choice possible.

Induction step $(\nu_M, l_M) \neq (1,1)$:

First case $l_M = 1$: Then $\nu_M > 1$ and there is an M' with

$$M \longrightarrow M',$$

$\nu_{M'} = \nu_M - 1$ and $(\nu_{M'}, l_{M'})$ precedes (ν_M, l_M) lexicographically. So $d_{M'}$ may be assumed to be defined.

$$d_M(i) =_{Df} \begin{cases} d_{M'}(i) & \text{for } i=1,\ldots,\nu_{M'} \\ \nu_M & \text{for } i=\nu_M \end{cases}$$

Second case $l_M > 1$: Then there is an M' with

$$M \dashrightarrow M',$$

$\nu_{M'} \le \nu_M, l_{M'} = l_{M'} - 1$ and $(\nu_{M'}, l_{M'})$ precedes (ν_M, l_M) lexicographically. So $d_{M'}$ may be assumed to be defined.

$$d_M(i) =_{Df} \begin{cases} d_{M'}(j) & \text{if } i \text{ is the } j\text{-th entry } \nu_{M_j} \text{ in the complement level list of } M, M' \\ \nu_{M'}+j & \text{if } i \text{ is the } j\text{-th number} \ge 1 \text{ not occurring in the complement level list of } M, M' \end{cases}$$

In case of ALGOL 60- or SIMULA 67-like programs with its same level prefixing we get the display register numbers association known from the literature. We easily prove by induction over the length of $\dashrightarrow$ -chains

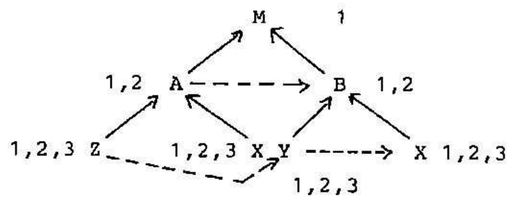
Lemma 4: The above mentioned condition is fulfilled and the complement level list of M, M' can be described by

$$\nu_{M_j} = d_M^{-1} \circ d_{M'}(j), \; j=1,\ldots,\nu_{M'},$$

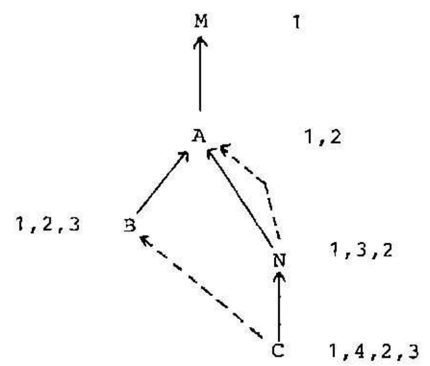
where d_M^{-1} is the inverse permutation of the numbers $i=1,\ldots,\nu_M$.

For the programs π_2, π_3 and π_4 of the Appendices C, E and F we have the following association:

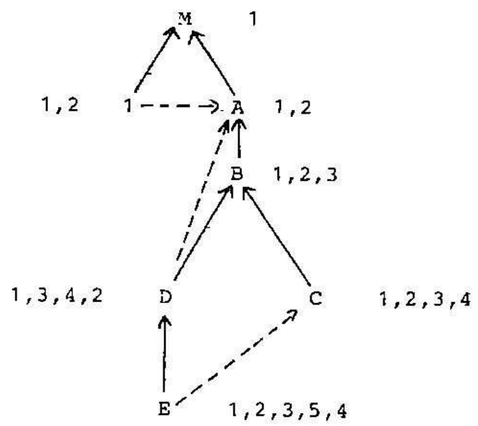
π_2 (SIMULA 67-like)



π_3



π_4



For program π_1 in Appendix B we have seen the association already in the introduction to part 2.

2.3 Design of the run time system for programs with many level prefixing

Let π be a distinguished proper program. Every module M has a certain fixed storage amount

$$fst(M) \geq 0$$

determined by the declarations of all variables ${}^{j}x$ with $env({}^{j}x)=M$. fst(M) is known at compile time. When M is activated then an activation record of the whole prefix chain of M

$$M=pref^{0}(M) \dashrightarrow pref^{1}(M) \dashrightarrow \dots \dashrightarrow pref^{l-1}(M)$$

of length $l>0$ is entered into the run time stack with a fixed storage amount

$$fstact(M)=K+fst(pref^{l-1}(M))+\dots+fst(pref(M))+fst(M)\geq 0.$$

K is a number > 0 known at compile time; K is the storage amount for the linkage of an activation record. So fstact(M) is known at compile time.

Let ${}^{j}x$ be a defining occurrence of a variable with $env({}^{j}x)=M$. The compiler reserves a storage cell in the fixed storage of M. This cell has a compile time known relative address or offset in the storage for the prefix chain of M. So

$$fstact(M)>reladdr(x)\geq fstact(M)-fst(M).$$

Let ${}^{i}x$ be an applied occurrence of a variable in the main part of module $M^{*}=env({}^{i}x)$ and let $bdfct({}^{i}x)={}^{j}x$ and $minmod(i,x)=\bar{M}$:

$$\begin{array}{ccc} \bar{M} & \overset{*}{\dashrightarrow} & M \\ {\scriptstyle *}\uparrow & & \\ M^{*} & & \end{array}$$

Let e.g. ${}^{i}x$ be the right hand side of an assignment statement

...:=x; ... .

Then we would like to compile this into a load instruction (assembler language)

LDA $d^{{}^{i}x}$, reladdr(${}^{j}x$)

which is read

"load accumulator from a cell
with address reladdr(${}^{j}x$) which
is modified (increased) by the
content of index (display) register
of number $d^{{}^{i}x}$".

In a higher level assembler language this instruction would look as follows

$\ldots\ldots := \mathfrak{M}[\vartheta[d^{i_x}] + reladdr(^{j}x)];\ldots$.

$\mathfrak{M}$ is a linear array representing the main storage of memory cells and ϑ is a linear array representing the series of index or display registers.

What display register number d^{i_x} do we take? We take

$$d^{i_x} =_{Df} d_{M^*}(\nu_{\bar{M}}) \ .$$

We demonstrate this definition for the compilation of the assignment statement

x:=y;

in class B of program π_1:

$\mathfrak{M}[\vartheta[2]+ reladdr(x)]:=\mathfrak{M}[\vartheta[2]+ reladdr(y)];$

and of

y:=x;

in class C:

$\mathfrak{M}[\vartheta[4]+ reladdr(y)]:=\mathfrak{M}[\vartheta[4]+ reladdr(x)];$.

Please remember: Display registers are to be loaded or reloaded only if a block ξ is entered or terminated, a procedure φ is called or terminated or a class η is initialialized by <u>new</u> η or terminated.

No reloadings shall happen when running through the main parts of the prefix chain of a module.

An activation record begins with K cells for linking with the following relative addresses and contents:

Relative address 0, mnemotechnically denoted $\equiv$ RA: <u>Return address</u> in the compiled program, where control has to go after regular termination.

Relative address 1 $\equiv$ DLD: <u>Dynamic level</u> of the <u>dynamic predecessor</u> of this activation record, i.e. the address of the return address cell of the immediately preceding activation record of this activation record.

Relative address 2 ≡ ID: Identifier of the activated block or procedure or initialized class.

Relative address 3 ≡ DLS: N cells for the dynamic levels of the static chain of this activation record. N≥1 is the maximal nesting level ν_M of all modules M in a program π.
If the activated module M has a level $\nu_M \geq 1$ then the first ν_M cells have relevant contents; the display registers $\vartheta[1],\ldots,\vartheta[\nu_M]$ are loaded resp. reloaded when this activation record is resp. becomes again the topmost entry of the run time stack and module M is activated resp. reactivated. The contents of the other $N-\nu_M$ cells are undefined.

Relative address N+3≡LG: Length of this activation record (relevant only for programs with global jumps).

So K is the number N+4 which is known at compile time.

The compiled program acts upon a run time stack in the main storage whibh is considered as an array

var $\mathfrak{M}$: array [0:∞] of something; .

The series of display registers forms an array

var ϑ: array [1:∞] of [0:∞]; .

The momentary dynamic level which shows to the return address cell of the momentary topmost activation record entry in the run time stack is held in a simple variable

var MDL: [0:∞]; .

The momentary free storage level is held in a simple variable

var FSL:[0:∞]; .

Further auxiliary variables

var AUX, AUX1 : [0:∞];

are used for procedure calls.

2.4 Compilation of essential program constructs

Let a distinguished proper program π be given such that w.r.o.g. every block has a block identifier.

I. A program π is a block

```
    η : block
      Δ
    begin
      Σ
    end η
```

and is compiled this way :

```
    call initialization;
    compiled Δ
    compiled Σ
    call finish program;
```

II. A non-prefixed block different from the whole program π

```
    η : block
      Δ
    begin
      Σ
    end η
```

with block module M_η is compiled this way[1] :

```
    call blockentering(η);
  Start 1 of Mη:
    compiled Δ
    compiled Σ
    call finish;
  End of Mη:
```

---

1) More efficient code will be generated if non-prefixed blocks η are treated like statements. They can be treated like special modules M_η without associated nesting level ν_{M_η}. Compilation is simply:

```
    compiled Δ
    compiled Σ.
```

III. A prefixed block

η : ξ block

Δ

begin

Σ

end η

with block module M_η is compiled this way: Let ξ denote a class with class module M_ξ. The translator reserves a variable x_ξ in the fixed storage of M_ξ:

call prefixed block entering(η);

Start 1 of M_η:

$\mathfrak{M}[\mathcal{V}[d_{M_\eta}(\nu_{M_\eta})]+\text{reladdr}(x_\xi)]$:= Start 2 of M_η;

goto Start 1 of M_ξ;

Start 2 of M_η:

compiled Δ

Start 3 of M_η:

$\mathfrak{M}[\mathcal{V}[d_{M_\eta}(\nu_{M_\eta})]+\text{reladdr}(x_\xi)]$:= Start 4 of M_η;

goto Start 3 of M_ξ;

Start 4 of M_η:

compiled Σ

goto After inner of M_ξ;

End of M_η:

Remember: x_ξ in block η is to be treated as an applied occurrence with its defining occurrence in class ξ. So minmod$(x_\xi)=M_\eta$ and the display register to be compiled is $d_{M_\eta}(\nu_{M_\eta})$ which is equal to $d_{M_\xi}(\nu_{M_\xi})$ due to Lemma 4. So it makes no difference whether we write $d_{M_\eta}(\nu_{M_\eta})$ or $d_{M_\xi}(\nu_{M_\xi})$.

IV. A non-prefixed class

    η : class
      Δ
    begin
      Σ_1 inner Σ_2
    end η;

with class module M_η is compiled this way:

The translator reserves a variable x_η in the fixed storage of M_η:

  Start 1 of M_η:
  Start 2 of M_η:
      compiled Δ
      goto $\mathfrak{M}[\vartheta[d_{M_\eta}(\nu_{M_\eta})]+\text{reladdr}(x_\eta)]$;
  Start 3 of M_η:
  Start 4 of M_η:
      compiled Σ_1
      goto $\mathfrak{M}[\vartheta[d_{M_\eta}(\nu_{M_\eta})]+\text{reladdr}(x_\eta)]$;
  After inner of M_η:
      compiled Σ_2
      call finish;
  End of M_η:

V. A prefixed class

    η : ξ class
      Δ
    begin
      Σ_1 inner Σ_2
    end η

with class module M_η is compiled this way:

  Start 1 of M_η:
      $\mathfrak{M}[\vartheta[d_{M_\eta}(\nu_{M_\eta})]+\text{reladdr}(x_\xi)]$:= Start 2 of M_η;
      goto Start 1 of M_ξ;
  Start 2 of M_η:
      compiled Δ
      goto $\mathfrak{M}[\vartheta[d_{M_\eta}(\nu_{M_\eta})]+\text{reladdr}(x_\eta)]$;
  Start 3 of M_η:
      $\mathfrak{M}[\vartheta[d_{M_\eta}(\nu_{M_\eta})]+\text{reladdr}(x_\xi)]$:= Start 4 of M_η;
      goto Start 3 of M_ξ;

Start 4 of M_η:
  compiled Σ_1
  goto $\mathfrak{M}[\mathfrak{v}[d_{M_\eta}(\nu_{M_\eta})]+\mathrm{reladdr}(x_\eta)]$;
After inner of M_η:
  compiled Σ_2
  goto After inner of M_ξ;
End of M_η:

VI. A non-prefixed procedure declaration

  φ : proc $(\xi_1,\ldots,\xi_n)$;
    Δ
  begin
    Σ
  end φ

with its procedure module M_φ is compiled this way:

Starting address of procedure φ:
  compiled Δ
  compiled Σ
  call finish;
End of M_φ:

VII. A prefixed procedure declaration

  φ : ξ proc $(\xi_1,\ldots,\xi_n)$;
    Δ
  begin
    Σ
  end φ

with its procedure module M_φ is compiled this way:

Starting address of procedure φ:
  $\mathfrak{M}[\mathfrak{v}[d_{M_\varphi}(\nu_{M_\varphi})]+\mathrm{reladdr}(x_\xi)]$:= Start 2 of M_φ;
  goto Start 1 of M_ξ;
Start 2 of M_φ:
  compiled Δ
Start 3 of M_φ:
  $\mathfrak{M}[\mathfrak{v}[d_{M_\varphi}(\nu_{M_\varphi})]+\mathrm{reladdr}(x_\xi)]$:= Start 4 of M_φ;
  goto Start 3 of M_ξ;

Start 4 of M_φ:
compiled Σ
goto After inner of M_ξ;
End of M_φ:

VIII. A non-formal procedure statement

call $\varphi(\alpha_1,\ldots,\alpha_n)$;

occurring in the main part of module M^* with module identifier χ is compiled this way (we assume that no actual parameter α_i invoces an implicit module activation):

compilation of α_1
$\mathfrak{M}$[FSL+reladdr(ξ_1)] := actual information about α_1;
⋮
compilation of α_n
$\mathfrak{M}$[FSL+reladdr(ξ_n)] := actual information about α_n;
compilation of call φ

$\xi_1,\ldots,\xi_n$ are the formal parameters of procedure φ corresponding to $\alpha_1,\ldots,\alpha_n$. Their storage cells are in the fixed storage of φ and their relative addresses are known at compile time.

Let ${}^i\varphi$ be the applied occurrence of φ immediately behind call. Let $\bar{M}$=minmod(i,φ) be the minimal module with $\bar{M} \xleftarrow{*} M^*$. If $d_{\bar{M}} = d_{M^*}[1:\nu_{\bar{M}}]$ and the module $M=M_\varphi$ to be entered is not prefixed then compilation of call φ is

call simple non-formal procedure(φ);
Return address of procedure call:

otherwise

call non-formal procedure(χ,φ);
Return address of procedure call:

In case the non-formal procedure statement

call $\varphi(\alpha_1,\ldots,\alpha_n)$

is only partially correct, but not correct then the statement is compiled into

error;

See the discussion about proper programs in Chapter 1.6.

IX. Let α_i be an actual parameter occurring in the main part of module M^* and let ξ_i be the corresponding formal parameter. What compiled code does compute the actual information about α_i? We have to differ between five cases IX.a to IX.e.

IX.a. Let α_i be a non-formal procedure identifier. Let $\bar{M}$ be minmod(α_i) with $\bar{M} \xleftarrow{*} M^*$. The actual information about α_i is a couple

(*) $(\alpha_i, \text{content of } \mathcal{V}[d_{M^*}(\nu_{\bar{M}})])$.

This information is computed by the code

$\alpha_i \oplus d_{M^*}(\nu_{\bar{M}})$

where $\oplus$ is a "machine operation" which couples the procedure identifier α_i with the content of display register $\mathcal{V}[d_{M^*}(\nu_{\bar{M}})]$ numbered by $d_{M^*}(\nu_{\bar{M}})$.

IX.b. Let α_i be a formal procedure identifier. The actual information about a couple like that above (*) is the content of

$\mathcal{M}[\mathcal{V}[d_{M^*}(\nu_{\bar{M}})]+\text{reladdr}(\alpha_i)]$,

and this is the code which computes the information (a load instruction with index register modification).

IX.c. Let α_i be an expression (e.g. of type real) and ξ_i be a formal input parameter (e.g. also of type real in order to avoid type transfers). The actual information about α_i is a real number computed by the compiled code of α_i.

IX.d. Let α_i be a non-formal simple variable (e.g. of type real) and ξ_i be a formal output variable (necessarily also of type real due to partial correctness). The actual information about α_i is an absolute address, namely the sum

$\mathcal{V}[d_{M^*}(\nu_{\bar{M}})]+\text{reladdr}(\alpha_i)$,

and this is the code which computes the information (a load and an add instruction).

IX.e. Let α_i be a formal output variable (e.g. of type real) and ξ_i be a formal output variable (necessarily also of type real). The actual information about α_i is an absolute address, namely the content of

$$\mathfrak{M}[\vartheta[d_{M^*}(\nu_{\bar{M}})]+\text{reladdr}(\alpha_i)]$$

and this is the code which computes the information (a load instruction with index register modification, compare IX.b.).

X. A class initialization statement

new η;

occurring in the main part of module M^* with module identifier χ is compiled this way:

compilation of init η

Start 1 of new:

$\mathfrak{M}[\vartheta[d_{M_\eta}(\nu_{M_\eta})]+\text{reladdr}(x_\eta)]$:= Start 2 of new;

goto Start 1 of M_η;

Start 2 of new:

Start 3 of new:

$\mathfrak{M}[\vartheta[d_{M_\eta}(\nu_{M_\eta})]+\text{reladdr}(x_\eta)]$:= After inner of M_η;

goto Start 3 of M_η;

Start 4 of new:

Return address of class initialization:

The compiler reserves a variable x_η in the fixed storage of the class module M_η.

Compilation of init η is done similar to call φ in VIII. and gives

call simple class(η);

respectively

call class(χ,η); .

XI. A formal procedure statement

call $\psi(\alpha_1,\dots,\alpha_n)$;

occurring in the main part of module M^* with its module identifier χ is compiled this way (we assume that no actual parameter α_i invoces an implicit module activation):

```
call prepare formal procedure(χ,ψ);
compilation of α_1
𝔐[AUX1+reladdr(ξ_1)] := actual information about α_1;
call check actual parameter(1);
      ⋮
compilation of α_n
𝔐[AUX1+reladdr(ξ_n)] := actual information about α_n;
call check actual parameter(n);
call formal procedure;
```

Return address of formal procedure call:

$\xi_1,\dots,\xi_n$ are the fictitious formal parameters of the formal procedure ψ corresponding to $\alpha_1,\dots,\alpha_n$. Their relative addresses are K-1+1,...,K-1+n, i.e. ψ is treated as if ψ had no prefix. Only in case ξ_i is a formal procedure

call check actual parameter(i);

needs to be compiled.

After this specification of code generation in section I. to XI. the subroutines

initialization,
blockentering(η),
finish,
prefixed blockentering(η),
simple non-formal procedure(φ),
non-formal procedure(χ,φ),
simple class(η),
class(χ,η),
prepare formal procedure(χ,ψ),
check actual parameter(i),
formal procedure

must be described. This will be done in Appendix G. A detailed proof of the correctness of this implementation will be given in a further publication.

Theorem 2: A proper Mini-LOGLAN-program π and its compiled program have the same state transformation rsp. input/output function as their semantics.

2.5. A run time system with short linkages

The run time system presented in chapter 2.3 and Appendix G is time efficient but space consuming because each linkage demands $N \geq 1$ cells to store dynamic levels of static chains. We want to make the linkages shorter. We reserve only one cell with relative address $3 \equiv DLS$ for the immediate static predecessor which is the content of the old $\mathfrak{M}$[dynamic level+DLS-1+ν_M-1] in case of the activated module M is $\neq M_1$. The new relative address for the activation record length is $4 \equiv LG$ and the new linkage length is $K \equiv 5$. If we successively go down the immediate static predecessors then we get the pseudo static chain of an activation record which the static chain is a part of.

The main problem for the reorganized run time system is to determine the static chain and the proper display registers loading when an activation record of a certain dynamic level dl is created or reactivated:

display registers loading : subroutine (dl:dynamic level);

begin

  if dl=0

  then $\mathcal{D}[1] := 0$

  else call display registers loading($\mathfrak{M}$[dl+DLS]);

     Let φ be the module identifier in cell $\mathfrak{M}$[dl+ID].

     Let $M' = strenv(M_\varphi)$ with $M_\varphi \longrightarrow M'$ and $\nu_{M'} = \nu_{M_\varphi} - 1$.

     We do the simultaneous assignment

$$\begin{pmatrix} \mathcal{D}[d_{M_\varphi}(1)] \\ \vdots \\ \mathcal{D}[d_{M_\varphi}(\nu_{M_\varphi}-1)] \\ \mathcal{D}[d_{M_\varphi}(\nu_{M_\varphi})] \end{pmatrix} := \begin{pmatrix} \mathcal{D}[d_{M'}(1)] \\ \vdots \\ \mathcal{D}[d_{M'}(\nu_{M'})] \\ dl \end{pmatrix}$$

  fi

end display registers reloading

Let us explain what the simultaneous assignment actually does: Let η be the module identifier in $\mathfrak{M}[\mathfrak{M}[dl+DLS]+ID]$. The preceding display registers reloading has given us the static chain of $\mathfrak{M}[dl+DLS]$ in the form

$$\begin{pmatrix} \vartheta[d_{M_\eta}(1)] \\ \vdots \\ \vartheta[d_{M_\eta}(\nu_{M_\eta}-1)] \\ \vartheta[d_{M_\eta}(\nu_{M_\eta})] \end{pmatrix}.$$

M' is in the prefix chain of M_η: $M_\eta \overset{*}{\dashrightarrow} M'$ with $\nu_{M'} \le \nu_{M_\eta}$.

$d_{M_\eta}^{-1} \circ d_{M'}|[1:\nu_{M'}]$ represents the complement level list of M_η,M'.

The static chain of M' is a subchain of the static chain above, namely

$$\vartheta[d_{M_\eta}(d_{M_\eta}^{-1} \circ d_{M'}(i))] = \vartheta[d_{M'}(i)]$$

for $i=1,\ldots,\nu_{M'}$. If we add dl then we have the static chain of M_φ resp. dl which is stored in

$$\begin{pmatrix} \vartheta[d_{M_\varphi}(1)] \\ \vdots \\ \vartheta[d_{M_\varphi}(\nu_{M_\varphi}-1)] \\ \vartheta[d_{M_\varphi}(\nu_{M_\varphi})] \end{pmatrix}.$$

Essential changes for the reorganized run time system are necessary only for the subroutines finish and formal procedure. The simultaneous assignment in finish is replaced by

display registers reloading(MDL); ,

in formal procedure by

$\mathfrak{M}$[FSL+DLS] := AUX;
display registers reloading(FSL); .

It is easy to transform the recursive subroutine display registers reloading into a more efficient iterative one. Activation records created by

call blockentering,
call simple non-formal procedure or
call simple class

should be specially marked. When such activation of a module M is finished and module M'=strenv(M) is not in the prefix chain of any module $\bar{M}$ with $\nu_{\bar{M}} > \nu_M$, then display registers reloading (MDL) needs not to go down to dynamic level 0 but only to $\mathfrak{M}$[FSL+DLS].

Subroutine display register reloading in some sense passes the genesis of an activation record. We now present a subroutine which helps to reconstruct the history of an activation record step by step. The subroutine is written as a function DLSP in a LOGLAN-like style which computes the dynamic level of the <u>immediate static predecessor of an activation record</u> with dynamic level dl and with $\mathfrak{M}$[dl + ID]= φ with respect to a module identifier ξ with $M_\varphi \overset{*}{\dashrightarrow} M_\xi$. If $\xi = \varphi$ then DLSP(dl,ξ)= $\mathfrak{M}$[dl+DLS].

DLSP: <u>function</u> (dl: dynamic level, ξ : module identifier): dynamic level;
   <u>var</u> ξ': module identifier; i: integer;
<u>begin</u>
     result:= $\mathfrak{M}$[addr+DLS];
     ξ':= id   <u>where</u> M_{id}=strenv($M_{\mathfrak{M}[dl+ID]}$);
     <u>for</u> i:=2 <u>to</u> $\nu_{M_{\mathfrak{M}[dl+ID]}} - \nu_{compl(M_{\mathfrak{M}[dl+ID]}, M_\xi, strenv(M_\xi))}$
     <u>do</u> result:= DLSP(result,ξ');
        ξ':= id'   <u>where</u> $M_{id'}$= strenv($M_{\xi'}$)
     <u>od</u>
<u>end</u> DLSP

<u>Lemma:</u> Let dl be the dynamic level of an activation record with $\varphi = \mathfrak{M}$[dl+ID] and $\nu_{M_\varphi} > 1$. Let ξ be a module identifier such that $M_\varphi \overset{*}{\dashrightarrow} M_\xi$. Then DLSP(dl,ξ)=dl' such that there exists a $k \geq 0$ with $pref^k(M_{\mathfrak{M}[dl'+ID]})$=compl($M_\varphi$, M_ξ, strenv(M_ξ)), i.e. dl' is the immediate static predecessor of dl w.r.t. ξ .

Let $|dl|$ denote the length of the pseudo static chain of the activation record with dynamic level dl. If $|dl|=2$ then for every ξ with $M \overset{*}{\dashrightarrow} M_\xi$ $\nu_{M_\xi}=2$ holds. $\mathfrak{M}$[dl+DLS] is the dynamic level of the activation record of module M_1 with $\nu_{M_1} = 1$ and is returned as value of DLSP(dl,ξ) since M_1=compl(M_φ, M_ξ, M_1^1).

Let $|dl|>2$ and assume that for every dl" with $|dl"|<|dl|$ in the pseudo static chain of dl the lemma already holds.

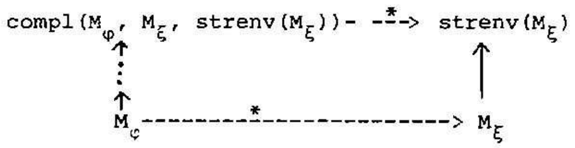
If $\varphi=\xi$ then k=0 and $M_{\mathfrak{M}[\mathfrak{M}[dl+DLS]+ID]} = compl(M_\varphi, M_\xi, strenv(M_\xi)) = strenv(M_\xi)$.

$\mathfrak{M}[dl+DLS]$ is returned as value of DLSP(dl, ξ) because the for-loop will not be executed.

If $\bar{\nu} =_{Df} \nu_{M_\varphi} - \nu_{compl(M_\varphi, M_\xi, strenv(M_\xi))} = 1$

then $strenv(M_\varphi) = compl(M_\varphi, M_\xi, strenv(M_\xi))$, i.e. there is a k=0 with $strenv(M_\xi) = pref^k(strenv(M_\varphi))$. Thus $\mathfrak{M}[dl+DLS]$ is the dynamic level of the immediate static predecessor of addr w.r.t. ξ as well as w.r.t. φ and is returned as value of DLSP(dl, ξ) because the for-loop will not be executed.

$\bar{\nu}\geq 2$: Considering the nesting tree of modules we have the following situation:



We have to compute the $\bar{\nu}$ static predecessor of dl w.r.t. "the path from M_φ to $compl(M_\varphi, M_\xi, strenv(M_\xi))$":

The first one is the immediate static predecessor of dl w.r.t. φ, thus it has the dynamic level $\mathfrak{M}[dl+DLS]$. Now for every of the $\bar{\nu}-1$ iterations of the for-loop we call DLSP with a dynamic level dl" of an activation record such that $|dl"|<|dl|$. The $\bar{\nu}$-th static predecessor is also the immediate static predecessor of dl w.r.t ξ and is returned as value of DLSP(dl, ξ).

Again essential changes for the runtime system are necessary only for the subroutine finish and formal procedure.

The simultaneous assignment in finish is replaced by

$\mathcal{V}[d_{M_\eta}(\nu_{M_\eta})]:=MDL;$

$\xi:=\mathfrak{M}[MDL + ID];$

for i:=ν_{M_η} -1 downto 2

do

  $\mathcal{V}[d_{M_\eta}(i)]:=DLSP(\mathcal{V}[d_{M_\eta}(i+1)],\xi);$

  $\xi := \xi'$ where $M_{\xi'}$ =strenv(M_ξ)

od;

in formal procedure by

$\mathfrak{M}[FSL + DLS]:= Aux;$

$\mathcal{V}[d_{M_\varphi}(\nu_{M_\varphi})]:=FSL;$

$\xi:=\mathfrak{M}[FSL + ID];$

for i:= ν_{M_φ} -1 downto 2

do

  $\mathcal{V}[d_{M_\varphi}(i):=DLSP(\mathcal{V}[d_{M_\varphi}(i+1)],\xi);$

  $\xi := \xi'$ where $M_{\xi'}$ =strenv(M_ξ)

od;

To load the required display registers only one call of subroutine display register loading is necessary whereas the user of DLSP must know how often it must be called. In both cases one has to go to the end of the pseudo static chain. Subroutine display register loading does operations on the display registers, i.e. intermediate results are held in them, when going back to the beginning of the pseudo static chain. DLSP holds its intermediate results in local variables. Operations on display registers must be done outside of DLSP when going to the end of the pseudo static chain.

Appendix A: A contextfree-like grammar for Mini LOGLAN.

```
<program>::=<block>
<block>::=
  {<block idf.>:}^01 <prefix class idf.>^01 block <body> <block idf.>^01
      |                   |                                   |
  redundant            applied                      redundant applied
  defining             occurrence                   occurrence, equal
  occurrence                                        to the matching
                                                    defining identifier

<body>::=<declaration list> begin <statement list> end
<declaration>::= <variable declaration>
               | <class declaration>
               | <procedure declaration>
<variable declaration>::= var <specification list>
<class declaration>::=
    <class idf.>: <prefix class idf.>^01 class <body> <class idf.>^01
        |               |                                  |
    defining        applied                  redundant applied
    occurrence      occurrence               occurrence, equal to
                                             the matching defining identifier
<procedure declaration>::=
   <procedure idf.>:<prefix class idf.>^01 proc<formal parameter list>;
          |        /  <body> <procedure idf.>^01
          |        \
     defining       applied          redundant applied occurrence,
     occurrence     occurrence       equal to the matching defining
<statement>::= <empty statement>     identifier
             |<error statement>
             |<assignment statement>
             | call <procedure idf> <actual parameter list>^01
                                   > applied occurrence
             | new <class idf.>
             | inner
             | <block>
             | <compound statement>
```

The superscript ^{01} is an indication that the superscripted entity may be there or not.

Appendix B: Program example π_1

```
M: block
   var x: real;
   A: class
       var x: real;
   begin
       x:=3;
       inner
   end A;
begin
     1: A block
        var y: real;
        B: class
        begin
            x:=y; print(x);
            inner
        end B;
      begin
          y:=2;
bdfct     2: new B;
          3: A block
             var y: real;
             C:3 class
             begin
                 y:=x; print(y);
                 inner
             end C;
          begin
              y:=4;
              4: new C
          end 3
      end 1
end M
```

Appendix C: Program Example π_2:

```
M: block
   B: class
      Y: X class
      begin
          u:= 3;
          inner
      end Y;
      X: class
         var u: real;
      begin
         inner
      end X;
   begin
      inner
   end B;
begin
   A: B block
      X: class
      begin
      end X;
   begin
      Z: Y block
      begin
      end Z
   end A
end M
```

prefix Y elimination

```
Z: X block
begin
    u:= 3;
end Z
```

u is a free identifier occurrence. If we would have renamed class X in block A into class X' then u would be bound to var u in class X in class B what is reasonable.

Appendix D:

Elimination of prefixes A in π_1 yields π_1' :

```
M: block
   var x: real;
                      (*  class A deleted *)
begin
    1: block
        var x: real;
        var y: real;
        B: class
        begin
            x:=y; print(x);
            inner
        end B;
    begin
        x:=3;
        y:=2;
        2: new B;
        3: block
           var x:  real;
           var y: real;
bdfct      C:B class
           begin
               y:=x; print(y);
               inner
           end C;
         begin
             x:=3; y:=4;
             4: new C
         end 3
    end 1
end M
```

Although in π_1 all applied occurrences of x have the same defining occurrence they have different ones in π_1'. This is so because they have different minimal modules (minmod) in π_1, and this has an influence when prefixes A in π_1 are eliminated.

Elimination of prefix B and new B in π_1' yields π_1'':

```
M: block
   var x: real;
begin
    1: block
       var x: real;
       var y: real;      ( * class B deleted  *)
   begin
       x:=3; y:=2;
       2: block
       begin
           x:=y; print(x);
       end 2;
       3: block
           var x: real;
           var y: real;
           C : class
           begin
               x:=y; print(x);
               y:=x; print(y);
               inner
           end C;
       begin
           x:=3; y:=4;
           4: new C
       end 3
   end 1
end M
```

Binding in case of dynamic scoping (without any renaming) is shown by arrows like ⟶ . Binding when only the source program π_1 is made distinguished (x in mainpart of block M and y in block 3 are renamed to $\bar{x}$ and $\bar{y}$) is shown by a correcting arrow --> . Binding in case of pure static scoping (all programs are made distinguished, especially all x in block 3 in π_1' are renamed to x') is shown by a correcting arrow ···>.

Elimination of new C in π_1'' yields π_1''':

```
M: block
   var x: real;
begin
    1: block
       var x: real;
       var y: real;
      begin
          x:=3;
          y:=2;
          2: block
          begin
              x:=y; print(x);
          end 2;
          3: block
             var x:real;
             var y: real;
                                  (* class C deleted *)
          begin
              x:=3; y:=4;
              4: block
              begin
                  x:=y; print(x);
                  y:=x; print(y);
              end 4
          end 3
    end 1
end M
```

Output of the program:

Dynamic scoping (—>): 2.0, 4.0, 4.0

Quasi-static scoping (--->): 2.0, 2.0, 2.0

Pure static scoping (··>): 2.0, 2.0, 3.0

Appendix E: Program example π_3:

```
      M: block
         var y:real;
         A:class
           var x:real;
           B:class
           begin
               x:=y;
               inner
           end B;
         begin
ΔM             1: new B;
             inner;
             N:A block
               var ȳ:real;
               C:B class
               begin
                   ȳ:=x;
                   inner
               end C;
             begin
                 2: new C
             end N
         end A;
      begin
          3: new A
      end M
```

Elimination of new A in π_3 yields π_3':

```
M: block
   Δ_M
begin
    3:block
        ⎧ var x: real;
        ⎪ B: class
        ⎪ begin
   Δ_3  ⎨     x:=y;
        ⎪     inner
        ⎩ end B;
    begin
        1: new B; ;
        ⎧ N: A block
        ⎪    var ȳ:real;
        ⎪    C:B class
        ⎪    begin
   Σ_N  ⎨        ȳ:=x;
        ⎪        inner
        ⎪    end C;
        ⎪ begin
        ⎪     2: new C
        ⎩ end N
    end 3
end M
```

Elimination of new B and prefix A in π_3' yields π_3'':

```
M: block
     Δ_M
begin
     3: block
          Δ_3
     begin
          1: block
          begin
               x:=y;
          end 1; ;
          N: block
             var x: real;
             B: class
             begin
                  x:=y; inner
             end B;
             var ȳ: real;
             C: B class
             begin
                  ȳ:=x; inner
             end C;
          begin
               1: new B;
               2: new C;
         ⎧     N: A block
         ⎪        var ȳ: real;
         ⎪        C:B class
  Σ_N    ⎨        begin
         ⎪             ȳ:=x; inner
         ⎪        end C;
         ⎪     begin
         ⎪          2: new C
         ⎩     end N
          end N
     end 3
end M
```

Repetition of Σ_N in π_3' and π_3'' shows that prefix elimination will never come to an end, the formal execution lattice E_{π_3} is infinite.

Appendix F : Transformation of π_1 yields π_1^T :

```
M : block
  var x : real;
  A : proc (A_f:proc(output real));
    var x : real;
  begin
    x := 3;
    call A_f(x)
  end A;
begin
  1 : block
    1_g : proc (output x:real);
      var y : real;
      B : proc (B_f:proc);
      begin
        x := y; print(x);
        call B_f
      end B;
    begin
      y := 2;
      2 : block
        2_g : proc;
        begin end 2_g;
      begin
        call B(2_g)
      end 2;
      Σ_3
    end 1_g;
  begin
    call A(1_g)
  end 1
end M
```

where Σ_3 is the following block :

```
3 : block
  3g : proc ( output x:real);
   var y : real;
   C : proc ( Cf:proc);
     Cg : proc;
     begin
       y := x; print(y);
       call Cf.
     end Cg;
   begin
     call B(Cg)
   end C;
  begin
   y := 4;
   4 : block
     4g : proc;
     begin end 4g;
   begin
     call C(4g)
   end 4
  end 3g;
begin
  call A(3g)
end 3
```

Transformation of π_3 yields π_3^T :

```
M : block
  var y real;
  A : proc(Af:proc(output real;proc));
    var x : real;
    B : proc (Bf:proc);
    begin
      x := y;
      call Bf
    end B;
  begin
    1 : block
      1g : proc;
      begin end 1g;
    begin
      call B(1g)
    end 1;
    call Af(x,B);
    ΣN
  end A;
begin
  3 : block
    3g : proc (output x:real;proc B(proc));
    begin end 3g;
  begin
    call A(3g)
  end 3
end M
```

where Σ_N is the following block :

```
N : block
  Ng : proc (output x:real;B:proc(proc));
    var ȳ : real;
    C : proc (Cf:proc);
      Cg : proc;
      begin
        ȳ := x;
        call Cf
      end Cg;
    begin
      call B(Cg)
    end C;
  begin
    2 : block
      2g : proc;
      begin end 2g;
    begin
      call C(2g)
    end 2
  end Ng;
begin
  call A(Ng)
end N
```

<u>Appendix G:</u> Run time system subroutines

initialization : <u>subroutine</u>;
<u>begin</u>
  MDL := 0;
  $\mathfrak{M}$[MDL+RA] := undefined;
  $\mathfrak{M}$[MDL+DLD] := undefined;
  $\mathfrak{M}$[MDL+ID] := identifier of the largest block of the program which also identifies the program;
  $\mathfrak{M}$[MDL+DLS-1+1] := ϑ[1] := 0;
  FSL := $\mathfrak{M}$[MDL+LG] := fstact(largest module M_1 of the program)
<u>end</u> initialization

blockentering : <u>subroutine</u> (η:blockidentifier);
<u>begin</u>
  $\mathfrak{M}$[FSL+RA] := program address for continuation when block η has regularly terminated, the address End of M_η is determined over the actual blockidentifier η;
  $\mathfrak{M}$[FSL+DLD] := MDL;
  $\mathfrak{M}$[FSL+ID] := η;
  We do the simultaneous assignment

$$\begin{pmatrix} \mathfrak{M}[\text{FSL+DLS-1+1}] \\ \vdots \\ \mathfrak{M}[\text{FSL+DLS-1+}\nu_{M_\eta}-1] \end{pmatrix} := \begin{pmatrix} \vartheta[d_{M_\eta}(1)] \\ \\ \vartheta[d_{M_\eta}(\nu_{M_\eta}-1)] \end{pmatrix};$$

  $\mathfrak{M}$[FSL+DLS-1+ν_{M_η}] := ϑ[d_{M_η}(ν_{M_η})] := FSL;
  $\mathfrak{M}$[FSL+LG] := fstact(M_η);
  MDL := FSL;
  FSL := FSL+fstact(M_η)
<u>end</u> blockentering

finish : <u>subroutine</u>;

<u>begin</u>

FSL := MDL;

MDL := $\mathfrak{M}$[MDL+DLD];

The module identifier η in cell $\mathfrak{M}$[MDL+ID] determines the prefix chain into which we return.

We do the simultaneous assignment

$$\begin{pmatrix} \mathfrak{v}[d_{M_\eta}(1)] \\ \vdots \\ \mathfrak{v}[d_{M_\eta}(\nu_{M_\eta})] \end{pmatrix} := \begin{pmatrix} \mathfrak{M}[\mathrm{MDL+DLS-1+1}] \\ \vdots \\ \mathfrak{M}[\mathrm{MDL+DLS-1+}\nu_{M_\eta}] \end{pmatrix};$$

<u>goto</u> $\mathfrak{M}$[FSL+RA]

<u>end</u> finish

prefixed blockentering : <u>subroutine</u> (η:blockidentifier);

<u>begin</u>

$\mathfrak{M}$[FSL+RA] := program address for continuation when block η has regularly terminated, the address End of M_η is determined over the actual blockidentifier η;

$\mathfrak{M}$[FSL+DLD] := MDL;

$\mathfrak{M}$[FSL+ID] := η;

Let strenv(M_η)=M' with $M_\eta \longrightarrow M'$ and $\nu_{M'} = \nu_{M_\eta} - 1$.

We do the simultaneous assignment

$$\begin{pmatrix} \mathfrak{M}[\mathrm{FSL+DLS-1+1}] \\ \mathfrak{M}[\mathrm{FSL+DLS-1+2}] \\ \vdots \\ \mathfrak{M}[\mathrm{FSL+DLS-1+}\nu_{M_\eta}-1] \\ \mathfrak{M}[\mathrm{FSL+DLS-1+}\nu_{M_\eta}] \end{pmatrix} := \begin{pmatrix} \mathfrak{v}[d_{M_\eta}(1)] \\ \mathfrak{v}[d_{M_\eta}(2)] \\ \vdots \\ \mathfrak{v}[d_{M_\eta}(\nu_{M_\eta}-1)] \\ \mathfrak{v}[d_{M_\eta}(\nu_{M_\eta})] \end{pmatrix} := \begin{pmatrix} \mathfrak{v}[d_{M'}(1)] \\ \mathfrak{v}[d_{M'}(2)] \\ \vdots \\ \mathfrak{v}[d_{M'}(\nu_{M'})] \\ \mathrm{FSL} \end{pmatrix};$$

$\mathfrak{M}$[FSL+LG] := fstact(M_η);

MDL := FSL;

FSL := FSL+fstact(M_η)

<u>end</u> prefixed blockentering

The subroutine blockentering can be replaced by the subroutine prefixed block entering; but blockentering is more efficient because in case of a non-prefixed block η we have

$$d_{M'} = d_{M_\eta}|[1:\nu_{M'}] .$$

simple non-formal procedure : <u>subroutine</u> (φ:procedure identifier);

<u>begin</u>

$\mathcal{M}$[FSL+RA] := Return address of procedure call which is transmitted by the subroutine call;

$\mathcal{M}$[FSL+DLD] := MDL;

$\mathcal{M}$[FSL+ID] := φ;

Let strenv(M_φ)=M' with $M_\varphi \longrightarrow M'$ and $\nu_{M'}=\nu_{M_\varphi}-1$.

We do the simultaneous assignment

$$\begin{pmatrix} \mathcal{M}[FSL+DLS-1+1] \\ \vdots \\ \mathcal{M}[FSL+DLS-1+\nu_{M_\varphi}-1] \end{pmatrix} := \begin{pmatrix} \mathcal{A}[d_{M'}(1)] \\ \vdots \\ \mathcal{A}[d_{M'}(\nu_{M'})] \end{pmatrix};$$

$\mathcal{M}[FSL+DLS-1+\nu_{M_\varphi}] := \mathcal{A}[d_{M_\varphi}(\nu_{M_\varphi})] :=$ FSL;

$\mathcal{M}$[FSL+LG] := fstact(M_φ);

MDL := FSL;

FSL := FSL+fstact(M_φ);

<u>goto</u> Starting address of procedure φ

<u>end</u> simple non-formal procedure

non-formal procedure : <u>subroutine</u> (χ:module identifier, φ:procedure identifier);

<u>begin</u>

$\mathcal{M}$[FSL+RA] := Return address of procedure call which is transmitted by the subroutine call;

$\mathcal{M}$[FSL+DLD] := MDL;

$\mathcal{M}$[FSL+ID] := φ;

Let strenv(M_φ)=M' with $M_\varphi \longrightarrow M'$ and $\nu_{M'}=\nu_{M_\varphi}-1$.

We do the simultaneous assignment

$$\begin{pmatrix} \mathcal{A}[d_{M_\varphi}(1)] \\ \vdots \\ \mathcal{A}[d_{M_\varphi}(\nu_{M_\varphi}-1)] \\ \mathcal{A}[d_{M_\varphi}(\nu_{M_\varphi})] \end{pmatrix} := \begin{pmatrix} \mathcal{M}[FSL+DLS-1+1] \\ \vdots \\ \mathcal{M}[FSL+DLS-1+\nu_{M_\varphi}-1] \\ \mathcal{M}[FSL+DLS-1+\nu_{M_\varphi}] \end{pmatrix} := \begin{pmatrix} \mathcal{A}[d_{M_\varphi} \circ d_{\bar{M}}^{-1} \circ d_{M'}(1)] \\ \vdots \\ \mathcal{A}[d_{M_\varphi} \circ d_{\bar{M}}^{-1} \circ d_{M'}(\nu_{M'})] \\ FSL \end{pmatrix};$$

```
  𝔐[FSL+LG] := fstact(M_φ);
  MDL := FSL;
  FSL := FSL+fstact(M_φ);
  goto Starting address of procedure φ
end non-formal procedure
```

M^* and $\bar{M}$ are defined as in section 2.4.VIII. where non-formal procedure statements are compiled. The subroutine simple non-formal procedure can be replaced by non-formal procedure; but the first one is more efficient because we have $d_{\bar{M}}=d_{M^*}|[1:\nu_{\bar{M}}]$ and $d_{M'}=d_{M_\varphi}|[1:\nu_{M'}]$ since M_φ is not prefixed.

```
simple class : subroutine (η: class identifier);
begin
  𝔐[FSL+RA] := Return address of class initialization which is
               transmitted by the subroutine call;
  𝔐[FSL+DLD] := MDL;
  𝔐[FSL+ID] := η;
  Let strenv(M_η)=M' with M_η ⟶ M' and ν_M' = ν_M_η - 1.
  We do the simultaneous assignment
```

$$\begin{pmatrix} \mathfrak{M}[\text{FSL+DLS}-1+1] \\ \vdots \\ \mathfrak{M}[\text{FSL+DLS}-1+\nu_{M_\eta}-1] \end{pmatrix} := \begin{pmatrix} \mathcal{N}[d_{M'}(1)] \\ \vdots \\ \mathcal{N}[d_{M'}(\nu_{M'})] \end{pmatrix};$$

$$\mathfrak{M}[\text{FSL+DLS}-1+\nu_{M_\eta}] := \mathcal{N}[d_{M_\eta}(\nu_{M_\eta})] := \text{FSL};$$

```
  𝔐[FSL+LG] := fstact(M_η);
  MDL := FSL;
  FSL := FSL+fstact(M_η)
end simple class
```

class : <u>subroutine</u> (χ:module identifier,η:class identifier);

<u>begin</u>

$\mathcal{M}$[FSL+RA] := Return address of class initialization which is transmitted by the subroutine call;

$\mathcal{M}$[FSL+DLD] := MDL;

$\mathcal{M}$[FSL+ID] := η;

We do the simultaneous assignment

$$\begin{pmatrix} \mathcal{N}[d_{M_\eta}(1)] \\ \vdots \\ \mathcal{N}[d_{M_\eta}(\nu_{M_\eta}-1)] \\ \mathcal{N}[d_{M_\eta}(\nu_{M_\eta})] \end{pmatrix} := \begin{pmatrix} \mathcal{M}[\mathrm{FSL+DLS}-1+1] \\ \vdots \\ \mathcal{M}[\mathrm{FSL+DLS}-1+\nu_{M_\eta}-1] \\ \mathcal{M}[\mathrm{FSL+DLS}-1+\nu_{M_\eta}] \end{pmatrix} := \begin{pmatrix} \mathcal{N}[d_{M^*}\circ d_{\bar{M}}^{-1}\circ d_{M'}(1)] \\ \vdots \\ \mathcal{N}[d_{M^*}\circ d_{\bar{M}}^{-1}\circ d_{M'}(\nu_{M'})] \\ \mathrm{FSL} \end{pmatrix};$$

$\mathcal{M}$[FSL+LG] := fstact(M_η);

MDL := FSL;

FSL := FSL+fstact(M_η)

<u>end</u> class

prepare formal procedure : <u>subroutine</u> (χ:module identifier, ψ:formal procedure identifier);

<u>begin</u>

$\mathcal{M}$[FSL+DLD] := MDL;

Let ψ be a formal parameter of a procedure with its module $\bar{M}$.

Let χ be the identifier of module M^*.

$\mathcal{M}$[FSL+ID] := first component($\mathcal{M}[\mathcal{N}[d_{M^*}(\nu_{\bar{M}})]+\mathrm{reladdr}(\psi)]$);

AUX := second component($\mathcal{M}[\mathcal{N}[d_{M^*}(\nu_{\bar{M}})]+\mathrm{reladdr}(\psi)]$);

Let φ be the non-formal procedure identifier in $\mathcal{M}$[FSL+ID].

$\mathcal{M}$[FSL+LG] := fstact(M_φ);

AUX1 := FSL+fstact(M_φ)-fst(M_φ)-K

<u>end</u> prepare formal procedure

check actual parameter : subroutine (i:parameter number);
begin
The specification of the actual procedure identifier in the first component of $\mathfrak{M}$[AUX1+K-1+i] is checked against the specification of the i-th formal parameter of that procedure the identifier of which is in $\mathfrak{M}$[FSL+ID]. In case of incorrectness computation is erroneously aborted
end check actual parameter

formal procedure : subroutine;
begin
$\mathfrak{M}$[FSL+RA] := Return address of formal procedure call which is transmitted by the subroutine call;
The non-formal procedure identifier φ in $\mathfrak{M}$[FSL+ID] has a defining occurrence ${}^{j}\varphi$ with $M'=env({}^{j}\varphi)$ and $M'=strenv(M_{\varphi})$, $M_{\varphi} \longrightarrow M'$, $\nu_{M'}=\nu_{M_{\varphi}}-1$.

The module identifier η in $\mathfrak{M}$[AUX+ID] identifies a module M_{η} with $\nu_{M_{\varphi}}-1=\nu_{M'} \leq \nu_{M_{\eta}}$.

We do the simultaneous assignment

$$\begin{pmatrix} \mathfrak{N}[d_{M_{\varphi}}(1)] \\ \vdots \\ \mathfrak{N}[d_{M_{\varphi}}(\nu_{M_{\varphi}}-1)] \\ \mathfrak{N}[d_{M_{\varphi}}(\nu_{M_{\varphi}})] \end{pmatrix} := \begin{pmatrix} \mathfrak{M}[FSL+DLS-1+1] \\ \vdots \\ \mathfrak{M}[FSL+DLS-1+\nu_{M_{\varphi}}-1] \\ \mathfrak{M}[FSL+DLS-1+\nu_{M_{\varphi}}] \end{pmatrix} := \begin{pmatrix} \mathfrak{M}[AUX+DLS-1+d_{M_{\eta}}^{-1} \circ d_{M'}(1)] \\ \vdots \\ \mathfrak{M}[AUX+DLS-1+d_{M_{\eta}}^{-1} \circ d_{M'}(\nu_{M'})] \\ FSL \end{pmatrix};$$

MDL := FSL;
FSL := FSL+$\mathfrak{M}$[MDL+LG];
goto Starting address of procedure φ
end formal procedure

Appendix H: Program Example #4

```
M: block
   A: class
      var x: real;
   begin
      B: block
         C: class
         begin
             x:=0;
             inner
         end C;
      begin
         D: A block
         begin
             E: C block
             begin
                 X:=0
             end E
         end D
      end B;
      inner
   end A;
begin
    1: A block
    begin
    end 1
end M
```

[Lo83] LOGLAN-82 Report, Polish Scientific Publisher, Warsaw 1983

[Na63] Naur,P. et al., Revised Report on the Algorithmic Language ALGOL 60, Numer.Math.4, 1963, pp. 420-453

[Wa84] Warpechowski,M., An Algebraic Model for Proving Address Properties in Languages with Prefixing and Module Nesting, Manuscript, Institute of Informatics, University of Warsaw, 1984