Concatenation of Program Modules
An Algebraic Approach to the Semantic and Implementation Problems

Manfred Krause, Hans Langmaack
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel, Germany

Antoni Kreczmar, Marek Warpechowski
Instytut Informatyki, Uniwersytet Warszawski, Poland

Abstract

The paper studies the semantic and implementation problems of programming languages which allow module concatenation. Three known languages of that class are Simula-67, Smalltalk and Loglan. The structure of program modules is treated as an algebra. A concise set of algebraic axioms defining this structure is given. The addressing problem is formulated in algebraic terms. The identifier binding rule is reduced to the evaluation of terms in the algebra of modules. The normal form theorem solves the question of this evaluation. The results allow to develop two efficient updating algorithms going beyond standard Dijkstra's algorithm and relevant for this class of languages. The paper ends with the detailed implementation techniques. The correctness of this implementation is proved. All of this allow to construct a new family of running-systems for languages with module concatenation.

1. Introduction

Module concatenation is a universal construct applicable in many situations arising in programming, especially because of the important role which it plays in object oriented languages [Hor 83, Chapter 14]. The concept was invented by O.J.Dahl, B.Myhraug and K.Nygaard and for the first time was introduced in Simula-67 [Da 70] under the term known as prefixing. In prefixing of modules prefixed module is an extension of a prefixing one, i.e. all the entities declared in the prefixing module are inherited to the prefixed one while the actions of the prefixing module enclose the actions of the prefixed module. Such an extension mechnism allows to build up hierarchies of modules in a program what aids program designing and program maintenance. In fact, packages, data structures and data types can be hierarchically designed, stepwise refinement, top-down and bottom-up programming methods are easily implementable, problem-oriented languages may be defined in a language,etc.

On the other hand in order to make changes one can concentrate on what needs to be changed instead of what must be left alone.

Up to now prefixing of Simula-67 is still underestimated, as well as the whole language, perhaps because of the old fashioned syntax and too much strong dependence on Algol-60 concepts. Another reincarnation of module concatenation appeared in Smalltalk [Ing 79]. The concept is called there inheritance and brings more or less the same ideas as prefixing in Simula-67. The main difference consists however in the way of defining the semantics and the accepted philosophy of implementation. Namely, in Smalltalk inheritance rule is treated as a pure textual operation and such is the basic philosophy of its implementation. On the other hand prefixing in Simula-67 may be implemented in the standard Algol-like method, i.e. with SL-links or Display vector, giving the static binding rule for identifier occurrences. This philosophy of implementation enables to define the semantics of the language which does not posses the well-known deviations appearing in any kind of textual or dynamic semantics. But to avoid these deviations the Simula designers had to bear the consequences, namely they had to restrict substantially the use of prefixing. To understand this restriction observe that when module concatenation is introduced in a block structured programming language, then there are two module operations: nesting (decl) and concatenating (pref). All the modules in a program create a tree with respect to module nesting and a module level is the depth of a module in such a tree. Simula restriction says that the prefixing of modules is allowed only if a prefixed and a prefixing module have the same level. This restriction is unnatural, diminishes the power of a language, unables the separate compilation etc. Even the Simula group has felt just from the starting point that the restriction should be overpassed and later on they initiated some works, e.g. [Kr 79], without satisfactory achievments.

In 1977 at the Institute of Informatics of Warsaw University the programming language Loglan was designed [Log 83]. This object oriented language includes the concatenation operation on modules without the Simula restriction. The solution proposed by A.Kreczmar [Ba 83] brought a new Display mechanism and a way of addressing giving the implementation as efficient as in Algol-60. In 1983 H.Langmaack [Kr 84] noticed that the implemented semantics is not purely static, i.e. that in some cases the semantics possesses the deviations which appear in textual or dynamic implementations. Later on H.Langmaack found a method of permuting Display addresses which preserves the rules of purely static semantics [Kr 84], however there was still not known how to update Display. In 1984 M.Krause invented a recursive algorithm updating properly Display and, simultaneously M.Warpechowski proposed a better iterative one. Moreover M.Warpe-

chowski proposed a new algebraic model for proving correctness of these algorithms [War 84], improved later by A.Kreczmar. The exposition presented in this paper bases on all these results.

The main problem of addressing can be factorized into the following two questions:

(1) give the way of finding at compile time for any applied occurrence of an identifier the corresponding declarative one (compile-time binding)

(2) give the way of finding at run time for any access to an entity its proper activation record and its offset (run-time binding)

The first question is trivial while the latter one, in the case of a language with module concatenation, becomes a research problem. Why this first one is trivial? Note that the inheritance rule for module concatenation says that inherited entities are local in a module. In consequence, when an instance of a module is generated, everything it inherits should be also generated, because it is local. Thus compile-time binding must give the preference for concatenation before nesting. For an applied occurrence of an identifier in a program we look first for its declaration in the most enclosing module. When not declared, we look for it in the prefix, if prefix is present. When all the consecutive prefixes are searched for, only then we look for the next enclosing module, and so on. Thus question (1) is a simple natural generalization of the well-known mechanism from Algol or Pascal.

Question (2) is trivial when we deal with a local entity. In fact, by the inheritance rule we can allocate at run time inherited instances and the instance itself as a one memory frame. The activation record for a local entity is so the part of this big frame and its relative displacement may be computed at compile time. Thus computing offsets we can take into account those relative displacements of the corresponding activation records what solves question (2) in the case of a local entity.

Question (2) in the case of a non-local entity may be reduced to the previous one, if we are able to compute the address of a corresponding frame. It turns out that the best way is to compute all such needed addresses before a given instance becomes active, and to keep them in an array called Display. This step reminds standard Dijkstra's approach. However the actions defined by prefixed instance involve the actions defined in the prefix. This may require the change of environment because compile-time binding concerns another module. In order to avoid this updating inside the concatenated module, a special permutation of Display array is needed. This method was proposed by H.Langmaack [Kr 84], as it was already mentioned.

The definition of this permutation as well as the correctness proof of the method begot the notion of a complementing module. Quite unexpectedly this notion turned out to be crucial for the solution of the hardest problem, i.e. Display updating algorithm, the problem which has not yet been solved when H.Langmaack defined his permutation. One attempt was initiated by M.Krause who wanted to reconstruct Display level by level, starting from the activated instance. This approach leads to a recursive algorithm whose implementation seems to be rather inefficient (we deal with a non-linear double recursion). Finally M.Warpechowski observed that the reconstruction of Display may be performed for each level independently what gives an iterative algorithm solving run-time binding problem in so efficient way as it is done in languages without module concatenation.

The structure of the paper is as follows. In section 2 we introduce a simple algebraic formalism. In section 3 the definition of an algebra of program modules is given, with two operations decl and pref. The notion of compile-time binding is precised and complementing module is defined. In section 4 run-time algebras are considered, corresponding to the given static algebra. Section 5 presents two updating algorithms, the first one is that of M.Krause, the latter one is that of M.Warpechowski, both with the correctness proof. In section 6 the Display array permutation is defined and proved to be correct. Section 7 giving the details of the implementation ends the paper.

2. Preliminaries

Let A denote any set and let f, g, ... denote partial functions from A to A. Next denote by $\perp$ an undefined element, i.e. $f(\perp)=\perp$ and $f(a)=\perp$ iff $f(a)$ is undefined.

Let $B = \{f_1,\ldots,f_n\}$ be a finite set of partial functions defined as previously and consider any word w written over alphabet B. Such a word designates the sequence of partial functions from set B that may be applied to an element of A in a given order. So, if a belongs to A and w is such a word, then the term $w(a)$ defines the given superposition of partial functions applied to a. According to this notation $w(a)=b$ where $b=\perp$ or b belongs to A, means that b is equal to the value designated by term $w(a)$.

Now let E be any regular expression over B, i.e. an expression built from $f_1,\ldots,f_n$ with the use of operators $\cup$, $\cdot$, $*$. Then by $E(a)$ we shall denote the set of terms:

$$E(a) = \{\, w(a) : w \in E \,\} .$$

For a from A and $b=\perp$ or b from A we shall write $E(a)=b$ iff there is a term $w(a)$ from $E(a)$ such that $w(a)=b$. Thus, for instance, $f^*(a)=b$ means that

there is $i \geq 0$ such that $f^i(a)=b$. Similarly, $f^+(a)=b$ means that there is $i>0$ such that $f^i(a)=b$. The equation $f^+(a)=\perp$ express the fact that there is an iteration of f such that when applied to a, the resulting value is undefined. With the use of this notation we can express much more complicated dependencies. For example, $f^*g^+(a)=b$ denotes that there are $i \geq 0$ and $j>0$ such that $f^i g^j(a)=b$, while $(f \cup g)^*(a)=b$ denotes that b may be obtained as a superposition of f and g, starting from a.

3. L-algebra

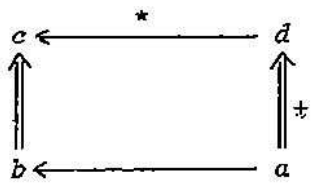
An L-algebra is an algebra $L = \langle M, decl, pref, p \rangle$ where M is a non-empty finite set, decl and pref are partial functions defined on M, p is an element of M and the following axioms are satisfied:

(A1) $decl(p)=\perp$ and for every a from M $decl^*(a)=p$,

(A2) for every a from M, $pref^+(a)=\perp$,

(A3) for every a from M, if $pref(a) \neq \perp$, then $decl\ pref(a) \neq \perp$ and $pref^* decl^+(a) = decl\ pref(a)$.

Axiom (A1) says that an algebra $\langle M, decl, p \rangle$ is a tree with the root p. Axiom (A2) says that an algebra $\langle M, pref \rangle$ is a forest. Axiom (A3) expresses the dependence between functions decl and pref. This dependence may be illustrated by the following diagram:



where $\longrightarrow$ denotes pref and $\Longrightarrow$ denotes decl. The existence of d such that the diagram commutes follows just from axiom (A3).

Definition 1. For any a from M by the level of a denoted by level(a) we shall mean $\min \{k: decl^k(a)=\perp\}$.

The existence of k such that $decl^k(a)=\perp$ follows from axiom (A1).

Definition 2. For a from M by the prefix sequence of a denoted by ps(a) we shall mean a sequence $(a_1,\ldots,a_n)$ where $pref(a_1)=\perp$, and $a_{n-i} = pref^i(a)$, for $i = n-1,\ldots,1,0$.

Definition 2 is correct because the existence of $i>0$ such that $pref^i(a)=\perp$ follows from axiom (A2).

Definition 3. For any two elements a, b from M by an address of b with respect to a denoted by address(a,b) we shall mean a pair (i,j) such that $j = \min\{k: pref^* decl^k(a)=b\}$, $i = \min\{k: pref^k decl^j(a)=b\}$, if $pref^* decl^*(a)=b$. Otherwise address(a,b) is undefined.

Observe that for $pref(a)\neq\perp$ address$(a,$ decl pref$(a))$ always exists by axiom (A3).

Lemma 1. $pref(p)=\perp$.

Proof

If $pref(p)\neq\perp$, then by (A3) decl pref$(p)\neq\perp$ and $pref^{*}decl^{+}(p)$ = decl pref(p). But by (A1) $decl(p)=\perp$ what implies that $decl^{+}(p)=\perp$. Hence we obtain decl pref$(p)=\perp$.

□

Lemma 2. For any two a,b from M, if $pref^{*}(a)=b$, then $level(b)\leq level(a)$.

Proof

We shall prove the lemma by induction on $level(a)$.

Start.

For $level(a)=1$ we have $a=p$. So, if $pref^{*}(p)=b$, then by Lemma1 $b=p$ what gives $level(b)=1$.

Induction step.

Suppose that the lemma holds for any such that $level(a)<n$.

Let $level(a)=n$ and $pref^{i}(a)=b$. Now we shall prove this step by induction on i.

Start.

For i=0 we have $b=a$, so $level(a)=level(b)$.

Induction step.

Suppose that $pref^{i-1}(a)=c$. By the inner induction $level(c)\leq level(a)$. Now let $pref(c)=b$. By axiom (A3) $decl(b)\neq\perp$, so we can assume that $decl(b)=d$. Again by axiom (A3) d = decl pref(c) = $pref^{i'}decl^{j}(c)$ for $i'\geq 0$, $j>0$. Let $decl^{j}(c)=e$. Since $j>0$ we have immediately $level(e)<level(c)$. Hence $level(e)<level(c)\leq level(a)=n$. By the outer induction $pref^{*}(e)=d$ and $level(e)<n$ implies $level(d)\leq level(e)$. Now $level(b)=level(d)+1$, $level(d)\leq level(e)$, $level(e)<level(c)$, $level(c)\leq level(a)$ gives $level(b)\leq level(a)$.

□

Definition 4. Let w be any word written over the alphabet B= {decl, pref}. By $\|w\|$ we shall denote the length of w and by $\|w\|_{decl}$ we shall denote the number of decl appearing in w.

Lemma 3. If $b=w(a)\neq\perp$, then $level(a)-level(b)\geq\|w\|_{decl}$.

Proof

Induction on $\|w\|$. If $\|w\|=0$, then $a=b$ and $level(a)-level(b)=0$. Let w = decl w_1. Then $level(a)-level(decl\ w_1(a))$ = $level(a)-level(w_1(a))+1$. From the inductive assumption $level(a)-level(w_1(a))\geq\|w_1\|_{decl}$. So $level(a)-level(b)\geq\|w_1\|_{decl}+1\geq\|w\|_{decl}$. Let w = pref w_1. Then $level(a)-level(pref\ w_1(a))\geq level(a)-level(w_1(a))$ because by Lemma 2 $level(pref\ w_1(a))\geq level(w_1(a))$. Hence $level(a)-level(b)\geq\|w_1\|_{decl}=\|w\|_{decl}$.

□

Definition 5. Let w, u be two words written over the alphabet B={decl,pref} and let $w=w_1$ decl pref w_2. For terms $w(a)$ and $u(a)$, where $a\in M$, we shall write

(*) $\quad w(a) \longmapsto u(a)$

iff $u=w_1 pref^i decl^j w_2$ where the pair (i,j) is address($w_2(a)$, decl pref($w_2(a)$)).

Definition 5 gives for any $a\in M$ and w the way of transformation $w(a)$ into another equivalent form. In general $\longmapsto$ is stronger than functional equivalence of terms, i.e. if $w(a) \longmapsto u(a)$ then $w(a)=u(a)$ as function values. Let $\overset{*}{\longmapsto}$ denote the reflexive and transitive closure of $\longmapsto$. In this way we obtain a kind of Post algorithm. Each step of this algorithm pushes one decl to the right after its immediately following pref. We want to show now that this process is always finite (what is not true for an arbitrary Post algorithm).

Lemma 4. For $a\in M$ and any w the sequence $w_k(a)$ such that $w_0=w$ and $w_k(a) \longmapsto w_{k+1}(a)$, $k=0,1,\ldots$ must be finite.

Proof

Suppose the contrary, i.e. that there is $a\in M$ and w such that the sequence $w_k(a)$ is infinite. By the form of the rule (*) we have that

$$\|w_k\|_{decl} \leq \|w_{k+1}\|_{decl} .$$

If $\|w_k\|_{decl}$, $k=0,1,\ldots$ has an upper bound, then this sequence has an element $w_m(a)$ such that for $k\geq m$ we have

$$\|w_k\|_{decl} = \|w_m\|_{decl} .$$

It means that from $w_m(a)$ rule (*) transforms decl pref into $pref^i decl$ (the number of decl does not change). But then from $w_m(a)$ after a finite number of steps we always obtain $u(a)$ of the form $pref^* decl^*(a)$, and rule (*) is not applicable for $u(a)$. So $\|w_k\|_{decl}$ has no upper bound.

Now we can use Lemma 3. We have $level(a)-level(w_k(a)) \geq w_k$ decl and the difference $level(a)-level(w_k(a))$ has no lower bound. This is impossible because $level(a)$ is finite. □

Definition 6. We shall say that $w(a)$ is in the normal form if $pref^* decl^* = w$.

Theorem. (Normal form theorem)

For every $a\in M$ and w such that $w(a)\neq\perp$ there is a unique $u(a)$ in a normal form such that $w(a) \overset{*}{\longmapsto} u(a)$.

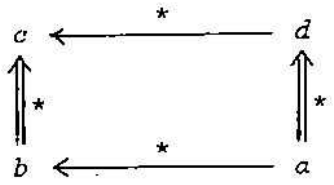
Proof

By Lemma 4 there is no infinite sequence of transformations according to the rule (*). Hence Post algorithm defined by these rules must always terminate. By Church-Rosser property the final result is unique indepen-

dently of the order of applications of (*). This final result must be in a normal form. □

Definition 7. Let $a, b, c \in M$ be such that $\mathrm{pref}^l(a)=b$ and $\mathrm{decl}^k(b)=c$. By a complementing element $\mathrm{compl}(a,b,c)$ we shall mean a unique element $d \in M$ such that $\mathrm{decl}^j(a)=d$ and $\mathrm{pref}^i(d)=c$ and $\mathrm{pref}^i\mathrm{decl}^j(a)$ is a normal form of $\mathrm{decl}^k\mathrm{pref}^l(a)$.

The correctness of Definition 7 follows from the normal form theorem. The diagram below illustrates the meaning of a complementing element:



The normal form theorem says that this diagram may be tiled in a unique way with the use of elementary tiles provided by axiom (A3).

Lemma 5. Let $\mathrm{pref}(a)=b$ and $\mathrm{pref}^+(b)=c$. Then

$$\mathrm{compl}(a,c,\mathrm{decl}^j(c)) = \mathrm{compl}(a,b,\mathrm{compl}(b,c,\mathrm{decl}^j(c))).$$

Proof

Let us consider the word of the form $\mathrm{decl}^j\mathrm{pref}^i\mathrm{pref}(a)$ where $\mathrm{pref}^i(b)=c$. Applying the normal form theorem we obtain:

$$\mathrm{decl}^j\mathrm{pref}^i(\mathrm{pref}(a)) = \mathrm{decl}^j\mathrm{pref}^i(b) \overset{*}{\longmapsto} \mathrm{pref}^*\mathrm{decl}^k(b)$$

where $\mathrm{decl}^k(b) = \mathrm{compl}(b,c,\mathrm{decl}^j(c))$. Applying again the normal form theorem we obtain:

$$\mathrm{decl}^k\mathrm{pref}(a) \overset{*}{\longmapsto} \mathrm{pref}^*\mathrm{decl}^l(a)$$

where $\mathrm{decl}^l(a) = \mathrm{compl}(a,b,\mathrm{decl}^k(b))$. Hence

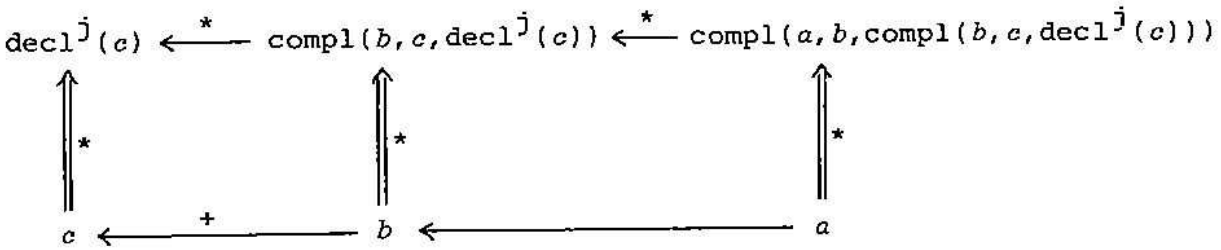
$$\mathrm{decl}^j\mathrm{pref}^i\mathrm{pref}(a) \overset{*}{\longmapsto} \mathrm{pref}^*\mathrm{decl}^k\mathrm{pref}(a) \overset{*}{\longmapsto} \mathrm{pref}^*\mathrm{decl}^l(a)$$

what implies that

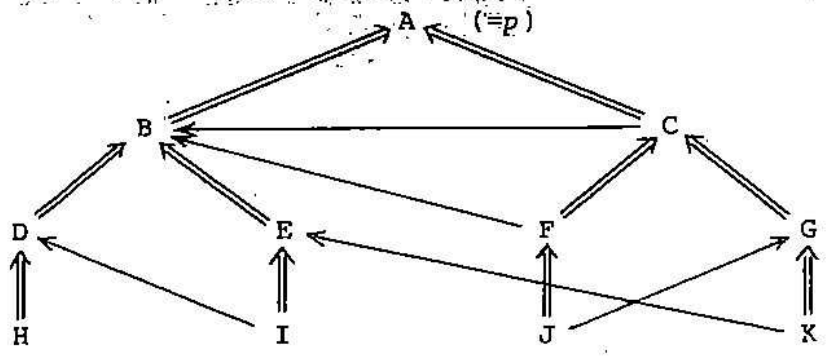
$$\mathrm{compl}(a,c,\mathrm{decl}^j(c)) = \mathrm{compl}(a,b,\mathrm{decl}^k(b)) = \mathrm{compl}(a,b,\mathrm{compl}(b,c,\mathrm{decl}^j(c))) .$$

□

The diagram below illustrates the meaning of Lemma 5.



Let us consider an example of an L-algebra. The diagram below illustrates its structure:



Here we have:

decl pref(C) = A = decl(C) decl pref(F) = A = decl²(F)
decl pref(I) = B = decl²(I) decl pref(K) = B = pref decl²(K)
decl pref(J) = C = decl²(J).

We can compute now the complementing elements for some triples:

compl(I,D,B)=B since B = decl pref(I) = decl²(I)
compl(F,B,A)=A since A = decl pref(F) = decl²(F)
compl(I,D,A)=A since A = decl²pref(I) = decl decl²(I) = decl³(I)
compl(J,G,A)=A since A = decl²pref(J) = decl decl²(J) = decl³(J)
compl(K,E,B)=C since B = decl pref(K) = pref decl²(K) = pref (C)
compl(J,G,C)=C since C = decl pref(J) = decl²(J) .

4. Implementations of L-algebra

Definition 1. The set of implementations IMP(L) of an L-algebra $L = \langle M, decl, pref, p\rangle$ is the smallest set of L-algebras $\overline{L} = \langle \overline{M}, decl, pref, \overline{p}\rangle$ embeddable into L and satisfying the following conditions:

(i) $\overline{L}_0 = \langle (\overline{p}), decl, pref, \overline{p}\rangle$ where $decl(\overline{p})=\bot$ and $pref(\overline{p})=\bot$ belongs to IMP(L),

(ii) For any $\overline{L}$ and $\overline{a}\in\overline{M}$ if $decl\ pref(\overline{a}) = \overline{c} \neq \bot$ then
$address(\overline{a},\overline{c}) = address(h(\overline{a}),h(\overline{c}))$
where $h:\overline{L}\to L$,

(iii) If $\overline{L}$ belongs to IMP(L), $h:\overline{L}\to L$, $\overline{b}\in\overline{M}$, $h(\overline{b})=b$ and $decl(a)=b$ for some $a\in M$, then there exists in IMP(L) an L-algebra $\overline{L}' = \langle \overline{M}', decl, pref, \overline{p}\rangle$ such that
$\overline{M}' = \overline{M} \cup \{\overline{a}_1,\dots,\overline{a}_m\}$, $ps(\overline{a})=(\overline{a}_1,\dots,\overline{a}_m)$, $\overline{a}_1,\dots,\overline{a}_m \notin \overline{M}$,
$\overline{L}'|_{\overline{M}} = \overline{L}$, $decl(\overline{a})=\overline{b}$,
$\overline{L}'$ is embeddable into L by homomorphism h' which extends h in such a way that $h'(\overline{a})=a$.

Lemma 1. Let a,b be from L and $\bar{a},\bar{b}$ be from $\bar{L}'$, and $\mathrm{decl}(a)=b$ as in Definition 1. Let $\mathrm{decl}(a_k)=b_k$ for $k=m,\ldots,1$ and let

$$\mathrm{address}(a_{k+1},b_k) = (i_k,j_k) \quad \text{for } k = m-1,\ldots,1 . \quad \text{Then}$$

$$\mathrm{decl}(\bar{a}_k) = \mathrm{pref}^{i_k}\mathrm{decl}^{j_k}(\bar{a}_{k+1}) \quad \text{for } k = m-1,\ldots,1 .$$

Proof

$$b_k = \mathrm{decl}\ \mathrm{pref}(a_{k+1}) \quad \text{and} \quad \mathrm{pref}(a_{k+1}) = a_k \neq \perp$$

So, by (A3) from section 3, we obtain

$$b_k = \mathrm{decl}\ \mathrm{pref}(a_{k+1}) = \mathrm{pref}^{*}\mathrm{decl}^{+}(a_{k+1}) .$$

Then, address (a_{k+1},b_k) is well defined. Homomorphism h' guarantees that

$$\mathrm{decl}\ \mathrm{pref}(\bar{a}_{k+1}) = \bar{b}_k \neq \perp .$$

Hence, by point (ii) of Definition 1

$$\mathrm{address}(\bar{a}_{k+1},\bar{b}_k) = \mathrm{address}(a_{k+1},b_k) = (i_k,j_k) .$$

And, from the definition of address (Definition 3, section 3)

$$\mathrm{decl}(\bar{a}_k) = \bar{b}_k = \mathrm{pref}^{i_k}\mathrm{decl}^{j_k}(\bar{a}_{k+1}) .$$

□

For any implementation L-algebra the elements of $\bar{M}$ will be called instances. They will be denoted by $\bar{a}$, $\bar{b}$, $\bar{c}$, etc. with indices, if necessary. The elements of M will be called modules. Moreover an instance $\bar{a}$ will be called an instance of a module a, if $h(\bar{a})=a$. For the sake of simplicity for any $\bar{a}\in\bar{M}$ the image $h(\bar{a})$ will be denoted by a, if it does not lead to any misunderstanding. So $\bar{a}$ denotes usually an instance of module a, if it is not especially stated.

Lemma 2. Let $\bar{L}$ be an implementation algebra of L. Then $w(a) \longmapsto u(a)$ in L implies $w(\bar{a}) \longmapsto u(\bar{a})$ in $\bar{L}$.

Proof

The proof follows immediately from Lemma 1, since the structure of rule (*) in Definition 5 section 3 remains the same.

□

The sequence of instances $(\bar{a}_1,\ldots,\bar{a}_m)$ introduced in the definition of $\bar{L}'$ (Definition 1) will be called an object. Objects will be denoted by capital letters X, Y, Z, with indices, if necessary. An instance $\bar{a}_m$ will be called the bottom instance of object $X=(\bar{a}_1,\ldots,\bar{a}_m)$.

When a function decl or pref is applied to an object X, then by default we assume it is applied to the bottom instance of X. For any instance $\bar{a}_i$ the object to which it belongs will be denoted by $|\bar{a}_i|$.

By Lemma 2 all the definitions concerning the normal form theorem and its applications are preserved in the implementation algebras. Hence the way of transformation the words of the form $w(\bar{a})$ in the implementation algebra

may be repeated in an algebra L giving the proper result.

Definition 2. Let Ob be the set of all objects for any given implementation algebra $\overline{L}$. The partial function SL: Ob → Ob is defined by the value of decl for the bottom instance of an object, namely:

if $|decl(X)| = Y$ then $SL(X) = Y$.

Lemma 3. For $X=(\overline{p})$ $SL(X)=\perp$ and for $X\neq(\overline{p})$ $pref^*(SL(X)) = decl(X)$.

Proof

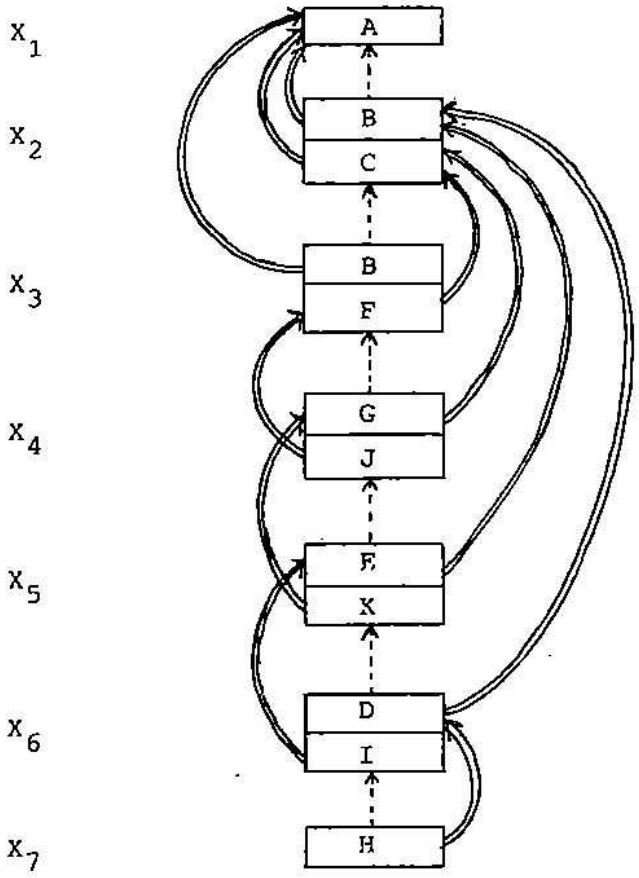
If $X=(\overline{p})$, then clearly $SL(X)=\perp$. Otherwise by Definition 2 we have $|decl(X)| = SL(X)$ what is equivalent to $pref^*(SL(X)) = decl(X)$. □

Lemma 4. For any object X, $SL^+(X)=\perp$.

Proof

We can prove the lemma by induction with the respect to Definition1. For $\overline{L}_0$ we have only one object $(\overline{p})$ and, of course, $SL((\overline{p}))=\perp$. Now consider an object $X=(a_1,\ldots,a_m)$ introduced in point (iii) of this definition. By the inductive assumption for $Y=SL(X)$ we have $SL^+(Y)=\perp$. Hence there is k such that $SL^k(Y)=\perp$. We have immediately $SL^+(X)=\perp$. □

For the example of an L-algebra given in section 3, below we give an illustration of one of its implementations:



where ⟹ denotes as usual decl, and -----➔ denotes SL.

5. Updating algorithms

The way of addressing in L-algebras must be copied in its implementations in order to have the so-called static scoping preserved. This was guaranteed by Definition 1 section 4 and Lemma 1 section 4. So the basic problem of addressing in the implementation algebras consists in the way we compute the address of the form $pref^i decl^j(\overline{a})$ for a given instance $\overline{a}$. However there is no problem with the computation of $pref^i$ since the number of iterations of pref may be established at compilation time and having given $decl^j(\overline{a})$ we can easily access an appropriate instance in the same object What is left out is the method of computation $decl^j(\overline{a})$ at run time. To solve this problem we shall start first with the simpler one which may be formulated in the following way. Having given an instance $\overline{a}$, $|\overline{a}| = X$, we search for an object Y such that $pref^*(Y) = decl(\overline{a})$. Observe, however, that $\overline{a} = pref^i(X)$ (we recall that X denotes the bottom instance of an object X as well). So we search for an object Y of the form:

$$|decl\ pref^i(X)| = Y \tag{1}$$

The easiest way is to compute at run time the function decl applied to X in the given order. But then we need to have one pointer for each instance in an object. We want to save space what may be done with the help of function SL. Hence we shall try to define the computation formulated by (1) in terms of function SL and functions decl and pref applied to algebra L instead of its implementation $\overline{L}$. To do this we shall apply first the normal form theorem obtaining from (1) the following term:

$$|decl\ pref^i(X)| = |pref^k decl^j(X)| = |decl^j(X)| = |decl^{j-1} decl(X)| \tag{2}$$

By Lemma 3 section 4 we have:

$$|decl\ pref^i(X)| = |decl^{j-1} pref^1(SL(X))| \tag{3}$$

Now for the termal form (3) we can apply (j-1) times the transformation defined by (1)-(3). If (j-1) = 0, then we are done, otherwise we call recursively the algorithm defined by this transformation. This recursive algorithm has stop property by Lemma 4 section 4. In fact, if $j > 1$, then the algorithm is applicable giving upon the exit the consecutive iteration of function SL. So finally there must be such a recursive call when upon the exit we obtain $j = 1$, since $SL^+(X) = \bot$. According to Lemma 2 section 4 all the computations of complementing modules may be done in algebra L.

The updating algorithm based on this observation was found by M.Krause. We shall present his algorithm in the form of a recursive function DLSP.

```
DLSP : function (X: object; a: module): object;
   {for any ā such that |ā| = X, the result is an object Y such that
    |decl(ā)|= Y }
  var e,f,c: module; i: integer;
  begin
    e:=h(X);                              {a = pref^i(e)  in (1)}
    f:=compl(e,a,decl(a));                {f = decl^j(e)  in (2)}
    X:=SL(X);  c:=decl(e);
    for i:=2 to level(e)-level(f)         { level(e)-level(f)=j in (2)}
    do
      X:=DLSP(X,e);  e:=decl(e)           { recursive call of DLSP for each
                                            (j-1)  decl in (3)}
    od;
    result:=X
  end DLSP;
```

Function DLSP solves the subproblem of the general updating problem which can be posed as follows. For a given object X_0 let us denote by D[1..lp] a display array (where lp = level(a), $|\bar{a}|$ = X), i.e. an array such that for k=1,...,lp D[k] = $|decl^{lp-k}(X_0)|$. So we have D[lp-1]= $|decl(X_0)|$, D[lp-2] = $|decl^2(X_0)|$, etc. Finally D[1] = $\bar{p}$ because $decl^{lp}(a) = p$. Array D shows during the execution of an object X_0 its syntactic environment.

The way we can compute the values of array D for any given object X_0 does not make now any difficulties. In fact, we can apply (lp-1) times function DLSP starting from instance X_0.

Algorithm K

```
  a:=h(X0);  lp:=level(a);  D[lp]:= X0;  X:=X0;
  for k:=lp-1  downto  1
  do
    X:=DLSP(X,a);  a:=decl(a);  D[k]:= X;
  od;
```

□

To illustrate Algorithm K we shall show how it works on algebra $\bar{L}$ presented as on the example in section 4. The only one interesting step is done when DLSP is called for X_6 and B, i.e. when D[2] = DLSP(X_6,B) is computed. We follow now this computation (the recursive calls are indented).

DLSP(X_6,B): e=I
f=compl(I,D,B)=B, X=X_5, c=E, level(e)-level(f)=4-2=2
for i=2 we execute the loop
DLSP(X_5,E): e=K
f=compl(K,E,B)=C, X=X_4, c=G, level(e)-level(f)=4-2=2
for i=2 we execute the loop
DLSP(X_4,G): e=J
f=compl(J,G,C)=C, X=X_3, c=F, level(e)-level(f)=4-2=2
for i=2 we execute the loop
DLSP(X_3,F): e=F
f=compl(F,F,C)=C, X=X_2, c=C, level(e)-level(f)=3-2=1
we do not execute the loop
result:=X_2
end of DLSP(X_3,F)
end of the loop in DLSP(X_4,G)
result:=X_2
end of DLSP(X_4,G)
end of the loop in DLSP(X_5,E)
result:=X_2
end of DLSP(X_5,E)
end of the loop in DLSP(X_6,B)
result:=X_2
end of DLSP(X_6,B).

Hence DLSP(X_6,B) = X_2 and D[2] = X_2. The execution of the complete Algorithm K in this case gives the following values of display array D, if the computation starts from object X_7:

$$D[1]= X_1 \quad D[2] = X_2 \quad D[3] = X_6 \quad D[4] = X_7.$$

A different approach to this updating problem was proposed by M.Warpechowski. He suggested to consider a modified problem. Namely, let $\overline{a}$ be the bottom instance of an object X and suppose we want to compute not only object |decl(X)|, but an arbitrary iteration of the form $|decl^k(X)|$. So we search for an object Y such that

$$|decl^k(X)| = Y \tag{4}$$

As before, we can try to transform the above term to an appropriate form. By Lemma 3 section 4 we have:

$$|decl^k(X)| = |decl^{k-1}decl(X))| = |decl^{k-1}pref^1(SL(X))| \tag{5}$$

By the normal form theorem from (5) we obtain:

$$|decl^{k-1}pref^{l}(SL(X))| = |pref^{i}decl^{k'}(SL(X))| = |decl^{k'}(SL(X))| \qquad (6)$$

Now, if $k'=0$ we are done. Otherwise we can iterate the process defined by transformations (4)-(6). According to Lemma 4 section 4, $SL^{+}(X) = \underline{\perp}$, so the algorithm must always terminate. This algorithm we shall present in the form of a non-recursive function WLSP.

```
WLSP : function (X: object; k: integer): object;
  {for a given object X the result is an object Y such that |decl^k(X)|=Y
var a,b: module;
begin
  a:=h(X);  b:=decl^k(a);                    {b̄ = decl^k(ā)  in (4)}
  while b≠a                                  {b = a  iff  k=0}
  do
    X:=SL(X);
    b:=compl(h(X),decl(a),b);                {b̄ is decl^k'(SL(X)) in (6)}
    a:=h(X)
  od;
  result:=X
end WLSP;
```

Now we shall try to apply function WLSP to the solution of the complete display updating problem. Observe first that (4)-(6) when applied to (k+1) instead of k give:

$$|decl^{k+1}(X)| = |decl\ pref^{i}decl^{k'}(SL(X))|$$

what after some number of steps reduces to:

$$|decl^{k+1}(X)| = |decl\ pref^{l}(SL^{r}(X)|.$$

In fact, this is the situation obtained upon the exit from function WLSP. But now we can apply again the normal form theorem, i.e. we can obtain:

$$|decl^{k+1}(X)| = |pref^{i'}decl^{j}(SL^{r}(X))| = |decl^{j}(SL^{r}(X))| \qquad (7)$$

what shows that the problem of computing $|decl^{k+1}(X)|$ may be reduced to the problem of computing $|decl^{k}(X)|$. So in the display updating algorithm we shall not call function WLSP and instead we shall use its main loop for the computation the successive elements of array D.

Algorithm W

```
a:=h(X0); lp:=level(a); D[lp]:=X0;
X:=X0;  e:=a;
for k:= lp-1 downto  1
do
  f:=decl(e);                          { f = decl^(lp-k)(h(X0))}
  b:=compl(a,e,f);                     { normal form presented  in (7)}
  while b≠a                            { function WLSP}
  do
    X:=SL(X);
    b:=compl(h(X),decl(a),b);
    a:=h(X)
  od;
  D[k] :=X;
  e := f
od;
```

□

For the example of algebra $\bar{L}$ given in section 4 we can illustrate the computations of Algorithm W in the following diagram:

	a	b	e	f	X	k	lp	
	H		H		X_7		4	$D[4]:=X_7$
		D = compl(H,H,D)		D		3		
WLSP {	I	I = compl(I,D,D)			X_6			$D[3]:=X_6$
		B = compl(I,D,B)	D	B		2		
	K	C = compl(K,E,B)			X_5			
	J	C = compl(J,G,C)			X_4			
WLSP {	F	C = compl(F,F,C)			X_3			
	C	C = compl(C,C,C)			X_2			$D[2]:=X_2$
		A = compl(C,B,A)	B	A		1		
WLSP {	A	A = compl(A,A,A)			X_1			$D[1]:=X_1$
			A					

6. Display registers enumeration

In the preceding section we solved the problem of finding for any object X a sequence $X_1,\ldots, X_{lp}$ such that $|decl^{lp-i}(X)| = X_i$, i=1,..., lp, where lp=level(a), $|\bar{a}|$=X. Hence having given this sequence kept in array D, we can easily compute the instances addressed from module a. Namely, for address(a,b)=(i,j) we should compute the instance $\bar{b} = pref^i decl^j(\bar{a})$ and $decl^j(\bar{a}) = decl^j(X) = decl^{lp-lp+j}(X) = X_{lp-j} = D[lp-j]$. But this is not the end of our possibilities.

Let us consider now a module c such that $pref^r(a)=c$ and let address(c,b) be (k,l). Then we want to find an instance $\bar{b}$ addressed from $\bar{c}$ in a similar way we have done it with an instance addressed from $\bar{a}$. Of course, we could repeat the whole process of searching for a new sequence $Y_1,\ldots,Y_{lp}$ defined by instance $\bar{c}$. But this is not necessary. In fact, we have:

$$\bar{b} = pref^k decl^l(\bar{c}) = pref^k decl^l pref^r(\bar{a})$$

and by the normal form theorem we obtain:

$$\bar{b} = pref^{i'} decl^{j'}(\bar{a})$$

what means that we can search for $\bar{b}$ in the sequence $X_1,\ldots, X_{lp}$ as it was defined for $\bar{a}$.

The above reasoning shows that the searching problem may be prepared uniformly for the whole prefix sequence $ps(a) = (a_1,\ldots,a_m)$. For each a_i and b such that $pref^* decl^*(a_i) = b$, we can search for $\bar{b}$ in the sequence $X_1,\ldots, X_{lp}$. This searching may be done even more efficiently if we introduce an appropriate enumeration of display items (the so-called display registers). Namely, suppose that for each module a there is defined a permutation of (1,..., lp) denoted by

$$dr(a) = (dr(a,1), dr(a,2),\ldots, dr(a,lp)).$$

When an object X, $\bar{a}$=X, is executed, display D[1..lp] should be defined in such a way that

$$D[dr(a,i)] = |decl^{lp-i}(X)| \quad \text{for } i=1,\ldots,lp \tag{1}$$

The way of defining such a permutation of display D is immediate, if we have prepared the permutation $dr(a)$ at compilation time. In fact, using any of the updating algorithms presented in section 5 we can assign $D[dr(a,i)]$ equal X_i, i=lp,...,1 , while X_i are computed as before. Suppose now that the permutations $dr(a)$ for $a \in M$ are defined so that the following condition is satisfied:

$$\text{if } pref^*(a)=b, \text{ then } dr(a,level(compl(a,b,decl^k(b)))) = dr(b,level(decl^k(b))) \tag{2}$$

Later on we shall show that such a definition of the permutations dr(a) is possible. The following lemma proves that this enumeration allows to access instances addressable from the prefix sequence of module a in the uniform way with the help of array D.

Lemma 1. If conditions (1) and (2) are satisfied, then for any module b such that $pref^*(a) = b$ with display D defined as for $|\bar{a}|=X$, we have:

$$|decl^k(\bar{b})| = D\,[dr(b, level(decl^k(b)))] \quad .$$

Proof

Let $compl(a,b,decl^k(b)) = decl^j(a)$. By Lemma 2 section 4 we have:

$$decl^j(\bar{a}) = compl(\bar{a},\bar{b},decl^k(\bar{b})).$$

Now by (2) :

$$D[dr(b,level(decl^k(b)))] = D[dr(a,level(decl^j(a)))]$$

and by (1):

$$|decl^j(\bar{a})| = |decl^{lp\text{-}level(decl^j(a))}(\bar{a})| = D[dr(a,level(decl^j(a)))].$$

Since $decl^j(\bar{a}) = compl(\bar{a},\bar{b},decl^k(\bar{b}))$ we have:

$$pref^* decl^j(\bar{a}) = decl^k(\bar{b})$$

what proves that:

$$|decl^k(\bar{b})| = D[dr(b,level(decl^k(b)))].$$

□

From Lemma1 we immediately see that the enumeration method satisfying (1) and (2) meets our needs. In fact, if display D is prepared as for $\bar{a}$, $pref^*(a)=b$, $pref^* decl^*(b)=e$ and $address(b,e)=(i,k)$, then we access the proper object since $|decl^k(b)|$ is equal to $D[dr(b,level(decl^k(b)))]$.

The last but not least thing which is left out is now the question whether we are able to define effectively the permutations dr(a) so that they satisfy condition (2). The idea of the construction is due to H.Langmaack, he formulated also the conditions (1) and (2). We shall present his construction in the form of a recursive procedure, however it can be easily reformulated in the form of the iterative algorithm. To simplify the presentation we assume that in a programming language, in which we define the algorithm, the catenation of one-dimensional arrays is admissible and denoted by &.

Algorithm L

```
DR : function (a: module): array[1..level(a)] of integer;
var k,j: integer;
begin
  if  a=p
  then
    result:=[1];                                  {one-element array}
    return
  else
    if pref(a) = ⊥
    then
      b:=decl(a);
      result:= DR(b) & [level(a)]  ;
      return
    else
      b:=pref(a);
      block
      var  help: array [1.. level(b)] of integer;
      begin
        help:= DR(b);
        for k:=1 to level(a)
        do
          result[k] := 0
        od;
        for k:=1 to  level(b)
        do
          result[level(compl(a,b,decl^k(b)))] := help[k]
        od;
        j:= level(b);
        for k:=1  to level(a)
        do
          if result[k] ≠0 then  j:=j+1 ; result[k] :=j  fi
        od
      end
    fi
  fi
end DR;
```

□

Lemma 2. Algorithm L is correct.

Proof

The correctness of function DR will be proved by induction with respect to level(a). If level(a)=1 , then dr(a)=(1) and function DR does the same. Suppose that for level(a) < n function DR produces correctly the permutations. Let us consider module a such that level(a)=n. If pref(a)= $\perp$, then condition (2) is trivially satisfied and we can simply extend dr(decl(a)) with the last value equal level(a) obtaining the permutation of the sequence (1,...,level(a)). Now suppose that pref(a)=b . By the definition of function DR we assign the values of dr(a) in such a way that condition (2) is satisfied by b. Finally consider module c such that $\text{pref}^i(b)=c$. By Lemma 5 section 3 with $\text{decl}^k(b)=\text{compl}(b,c,\text{decl}^j(c))$:

$$\text{compl}(a,c,\text{decl}^j(c))=\text{compl}(a,b,\text{compl}(b,c,\text{decl}^j(c))=\text{compl}(a,b,\text{decl}^k(b)) \qquad (3$$

We prove the correctness of function DR by induction on i. If i=0, then $c=b$ and condition (2) is satisfied by b. To prove the inductive step we have from the inductive assumption:

$$\text{dr}(b,\text{level}(\text{compl}(b,c,\text{decl}^j(c))))=\text{dr}(c,\text{level}(\text{decl}^j(c))) \qquad (4$$

and by the correctness of DR for pref(a)=b we obtain:

$$\text{dr}(a,\text{level}(\text{compl}(a,b,\text{decl}^k(b))))=\text{dr}(b,\text{level}(\text{decl}^k(b))) \qquad (5$$

From (3) and (5) we have:

$$\text{dr}(a,\text{level}(\text{compl}(a,c,\text{decl}^j(c))))=\text{dr}(a,\text{level}(\text{compl}(a,b,\text{decl}^k(b))))=$$
$$=\text{dr}(b,\text{level}(\text{decl}^k(b))).$$

But according to (3) $\text{decl}^k(b) = \text{compl}(b,c,\text{decl}^j(c))$, so we have:

$$\text{dr}(a,\text{level}(\text{compl}(a,c,\text{decl}^j(c))))=\text{dr}(b,\text{level}(\text{compl}(b,c,\text{decl}^j(c)))$$

and by (4) we obtain the thesis.

□

Let us compute the permutations dr(a) for all $a \in M$, where M is defined as in the example of L-algebra in section 3. We have:

dr(A)=(1) dr(B)=(1,2) dr(C)=(1,2) dr(D)=(1,2,3) dr(E)=(1,2,3)
dr(F)=(1,3,2) dr(G)=(1,2,3) dr(H)=(1,2,3,4) dr(I)=(1,2,4,3)
dr(J)=(1,2,4,3) dr(K)=(1,2,4,3).

7. Implementation of algorithms

We shall show now how to implement efficiently Algorithm W presented in section 5. This algorithm is evidently equivalent to the following one:

Algorithm W2

```
a:=h(X); k:=level(a); d:=a ;
do
  D[k]:= X;
  if k=1 then exit fi;
  k:=k-1;  b:=compl(a,d,decl(d));  d:=decl(d);
  do
    c:=decl(a); X:=SL(X);  a:=h(X);  b:=compl(a,c,b);
    if b=a then exit fi
  od
od;
```

□

Now instead of computing a module b we can compute its level. Let us introduce a new function ckl defined as follows:

$$ckl(a,c,j)=k \quad \text{iff} \quad level(compl(a,c,decl^{level(c)-j_{(c)}}))=k$$

In fact, function ckl gives for $pref^{*}(a)=c$ and $j=level(decl^{i}(c))$ the level k of the complementing module $compl(a,c,decl^{i}(c))$. With the help of function ckl the above algorithm W2 may be transformed into the equivalent algorithm W3:

Algorithm W3

```
a:=h(X);  k:=level(a); d:=a ;
do
  D[k] :=X;
  if k=1 then exit fi;
  k:=k-1 ;  j:=ckl(a,d,k);  d:=decl(d);
  do
    c:=decl(a);  X:=SL(X);  a:=h(X);  j:=ckl(a,c,j);
    if level(a)=j  then exit fi;
  od
od;
```

□

The main advantage of function ckl over compl is that it can be easily computed with the help of permutation $dr(a)$ introduced in the previous section.

Lemma 1. $ckl(a,b,j) = dr^{-1}(a,dr(b,j))$.

Proof

Suppose that $ckl(a,b,j)=level(a)-k$, where $j=level(b)-i$. It means that $compl(a,b,decl^i(b))=decl^k(a)$. But according to (2) section 6 we have:

$$dr(a,level(compl(a,b,decl^i(b))=dr(b,level(decl^i(b))=dr(b,j).$$

On the other hand:

$$level(compl(a,b,decl^i(b))=level(decl^k(a))=level(a)-k.$$

Hence $dr(a,level(decl^k(a))=dr(b,j)$ what implies that:

$$level(a)-k=dr^{-1}(a,dr(b,j)).$$

□

By Lemma 1 we see how simple is the updating algorithm. At run time we need only to have precomputed dr and dr^{-1} for each module a. The complete solution is presented in the following algorithm:

Algorithm Update

```
prototype : class;
   var decl,pref: prototype; level: integer;
       dr,drinv: array of integer;
   end prototype;
object: class;
   var pt: prototype; SL: object;
   end object;
var D:  array of object;
update: procedure(X: object);
   var a,c,d,e :-prototype; j,k: integer;
   begin
     a:=X.pt; k:= a.level;  d:=a ; e:=a ;
     do
        D [e.dr[k]] :=X;
        if k=1 then exit fi;
        k:=k-1 ;   j:= a.drinv [d.dr[k]];  d:= d.decl;
        do
         c:= a.decl;  X:=X.SL;  a:=X.pt;
           j:= a.drinv [ c. dr [j]] ;
           if a.level=j then exit fi;
        od
     od
   end update;
```

□

References

[Ba83] Bartol,W.M., Kreczmar,A., Litwiniuk,A.I., Oktaba,H., Semantics and Implementation of Prefixing at Many Levels, in: Logics of Programs and their Application, LNCS148, 1983,45-80

[Da70] Dahl,O.J., Myrhaug,B., Nygaard,K., Simula 67 Common Base Language, Norwegian Computer Center, 1970

[Hor83] Horowitz,E., Fundamentals of Programming Languages, Springer Verlag, 1983

[Ing79] Ingalls,D.H., The Smalltalk 76 Programming System Design and Implementation, Proc.5th ACM Principles of Prog.Lang.,1976,9-16

[Kr84] Krause,M., Kreczmar,A., Langmaack,H., Salwicki,A., Specification and Implementation Problems of Programming languages Proper for Hierarchical Data Types, Report 8410, Institut fuer Informatik und Praktische Mathematik, Universitat Kiel, 1984

[Kr79] Krogdahl,S., On the Implementation of Beta, Norwegian Computing Center, 1979

[Log83] Loglan-82 Programming Language,Report, Polish Scientific Publisher, Warsaw 1983

[War84] Warpechowski,M., An Algebraic Model for Proving Address Properties in Languages with Prefixing and Module Nesting,Manuscript, Institute of Informatics, University of Warsaw, 1984