On the correctness and completeness of a deterministic algorithm elaborating direct superclasses in Java-like languages

Hans Langmaack
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Andrzej Salwicki
National Institute of Telecomunication
Warsaw

Marek Warpechowski
Institute of Informatics, Warsaw University
{*salwicki* | *warp*} @*mimuw.edu.pl*

Abstract. In an earlier article [4] we analyzed the problem of determining direct superclasses in Java and Javalike languages. We gave a specification of the problem showing that it closely reflects the requirements of [2]. We presented a nondeterministic algorithm and proved its correctnnes and completeness. This paper presents a deterministic algorithm which elaborates direct superclasses as well as new problems. Another advantage of the proposed algorithm is error recovery. Should the algorithm report an error it continues its job and eventually reports more errors at one pass. A problem arises in connection with this feature of our algorithm: can we trust the subsequent error signals? is it possible to prepare better scheme of error recovery?

1. Introduction

Java allows inheritance of classes and inner classes. The combination of these two tools of developing programs proved its usefulness. cf.[4] and [1] – chapter on inner classes. On the other hand it complicates the life of programmers and of compiler writers. One of problems that appeared is the elaboration of direct superclasses. Let the header of a class K be **class** K **extends** B $\{$. Which of possibly many classes B is inherited? For there may be several classes of the same name in one program. Moreover Java allows to inherit from classes that are values of qualified types of the form $C_1.C_2.....C_n$ For example, a class K may begin "**class** K **extends** $C_1.C_2.....C_n$ $\{$". Now the problem of determining which class of name C_n is the direct superclass of class K is even more complicated. It may happen that qualified types used in a program define a cycle of mutually depending classes. Without solution of this problem one can not make the static semantic analysis of source code. Obviously compilers exist and programmers are using them. However, our experiments showed that compilers are not unanimous, it may happen that different compilers indicate direct superclasses in different way. There are also cases that different com-

pilers detect erroneous structures of classes and signal it improperly. Also programmers have different opinions and indicate direct super classes in different ways. The reference manual of Java: Java Language Specification is a thick volume and the needed information is scattered throughout it. One attempt to standardize the semantics was made by E. Boerger and his colleagues [5]. The book is restricted to Java 1.1 and does not discuss the problems caused by inner classes. A paper by Igarashi and Pierce [3] was the first to discuss the semantics of inner classes. In its section 5 we found the rules for elaboration of direct super classes. The set of rules together with the principle "apply the rules bottom up" was proposed by Igarashi and Pierce as a method of elaboration of types. The method has some drawbacks. The most important is that it does not guarantee the halting property [3]. Moreover the method has no means to signal eventual errors in extends clauses. In a recent paper we proposed a non-deterministic algorithm and made its analysis proving its correctness and completeness. The non-deterministic algorithm is an abstraction, some details of implementation are omitted. It does a diagnostics of erroneous source codes but has no error recovery mechanism. All these drawbacks are eliminated in the algorithm presented below. The algorithm works in the data structure of classes which is a simplification of the symbol table of a compiler. We present a formulation of the problem, a deterministic algorithm, an analysis of semantic properties of the algorithm. We show that the algorithm does precise diagnostic of possible errors in source code as well as error recovery which makes possible to continue collecting more errors after the first one was found.

2. Data structure of classes and the formulation of the problem

This section is a modification of section 2 of [4]. Let P be a Java program. In our discussion we shall use the structure of classes:

$$\mathcal{S} = \langle Classes, \{null\}, Id, Types, decl, name, ext, Root, Object \rangle$$

where

- $Classes$ is the set of classes declared (more distinctly: class declaration occurrences) in a program plus two predefined classes $Root$ and $Object$,
- Id is the set of identifiers (allowed by the programming language),
- $Types$ is the set of types, simple or qualified i.e. finite sequences of identifiers separated by dots, the empty sequence ε included,
- $decl : Classes \setminus \{Root\} \longrightarrow Classes$ is the function which for each class $K \neq Root$ returns the class textually, directly enclosing K. For simplicity we see $Object$ as a class without any classes declared inside. We assume $decl(Root) = null$.
- $name : Classes \setminus \{Root\} \longrightarrow Id$ is the function that returns the identifier of a given class. Class $Root$ has no name, $name(Object)$ = Object.
- $ext : Classes \setminus \{Root, Object\} \rightarrow Types$
  is the function which for each class $K \notin \{Root, Object\}$ returns the extension type. If the extension clause is omitted in the declaration of class K then $ext(K) = \varepsilon$. Otherwise $ext(K)$ is equal to the type found in the extension clause.

- $Root$ and $Object$ are distinguished elements of the set $Classes$.

The structure $\mathcal{S}$ has the following properties.

- $decl(Object) = Root$
- The pair $\langle Classes, decl\rangle$ is a tree. The class $Root$ is its root.
- If $decl(K) = decl(M)$ then $name(K) \neq name(M)$ or $K = M$

In the remainder of this paper we shall use the function $.C : Classes \to Classes \cup \{null\}$.
Let K be a class, let C be an identifier. The expression $K.C$ denotes a class Y which is declared within class K and its name is C or $null$ if there is no such a class. $K.C = Y$ if and only iff $(\exists Y)(decl(Y) = K \wedge C = name(Y))$. Otherwise $K.C = null$. The well-definedness of function $.C$ follows from the third property listed above. Given a structure $\mathcal{S}$ the problem is to obtain a partial function inh which for every given class $K \neq Root, K \neq Object$ returns the direct superclass of class K or to assure that such a function does not exist, signalling that the class structure $\mathcal{S}$ is not a correct one. In fact we are seeking two functions *inh* and *bind*. Their definitions are mutually involved. In the following we give an inductive definition of the partial function $bind(type \text{ in } class)$ which associates a *class* to a given pair $\langle type, class\rangle$. An equation $bind(T\ in\ C) = D$ reads informally as: the meaning of type T inside the class C is class D. Compare our definition with the text of Java Langugage Specification [2](6.3, p. 85, and follow references 6.5.5, 8.1.3 ...). We believe that our definition of the function $bind$ corresponds most closely to the lengthy and scattered description of meaning of Java's type name. In contradiction to the Java Language Specification, the paper [3] defines the meaning of type names even for some incorrect (non well-formed in the sense of [2]) programs. For details see section Elaboration of Types in [3].

Definition 2.1. (*base of induction* A) For any class K the meaning of the empty type ε is bound to Object. We define

$$bind(\varepsilon\ in\ K) \stackrel{df}{=} Object.$$

(*base of induction* B) Let K be a class. An applied occurrence of a (class) identifier C in the class K is *bound* to a class named C such that

$$bind(C \text{ in } K) \stackrel{df}{=} (inh^i decl^j(K)).C$$

where the pair (j, i), $j \geq 0$, $i \geq 0$, is the least pair in the lexicographic order such that the value $(inh^i decl^j(K)).C$ is $\neq null$. The pairs are compared according to the lexicographical order, i.e. the pair (j, i) is *less* than the pair (q, p) if $j < q$ or $j = q$ and $i < p$. The value of $bind(C \text{ in } K)$ is $null$ in the remaining cases.

(*inductive step* C). Let $X \neq \varepsilon$. For any class K the meaning of a type of the form $X.C$ in the class K is determined in two steps.

$$bind(X.C\ in\ K) \stackrel{df}{=} (inh^i(bind(X\ in\ K)).C$$

where $i \geq 0$, is the least integer such that the inequality $(inh^i(bind(X\ in\ K)).C \neq null$ holds, the value of $bind(X.C\ in\ K)$ is $null$ in the remaining cases.

The following relation dep plays an important rôle in the further considerations.

Definition 2.2. dep is a binary relation in the set of Classes such that

$$dep \stackrel{df}{=} \{\langle K, bind(ext(K)|^i \; in \; decl(K))\rangle : K \in Classes \setminus \{Root, Object\},$$
$$0 < i \leq length(ext(K)) \text{ for } ext(K) \neq \varepsilon,$$
$$i = 0 \text{ for } ext(K) = \varepsilon\}$$

where the expression $ext(K)|^i$ denotes the initial segment, of length i, of the type $ext(K)$.

Let $ext(K)$ be the following type $C_1.C_2.$ ... $.C_i.$ ... $.C_n$. Then $ext(K)|^i$ is the type: $C_1.C_2.$ ... $.C_i$. Figure 1 illustrates the way of computing the value of the function inh and the relation dep. Now we are ready to specify the problem of determining the direct superclasses.

Problem 2.1. For a given structure of classes $\mathcal{S}$, determine whether there exist

- a function $bind$ and a relation dep, defined as in the preceding definitions, and,
- a function $inh : Classes \setminus \{Root, Object\} \longrightarrow Classes \setminus \{Root\}$,

such that the following two conditions hold:

I_1) the values $inh(Root)$ and $inh(Object)$ are equal $null$ and, for every class $K \notin \{Root, Object\}$ the value $inh(K)$ is different than $null$ and the following equality holds

$$inh(K) = bind(ext(K) \text{ in } decl(K)),$$

I_2) the induced relation dep has no cycle.

and produce the function inh if it exists.

Definition 2.3. A structure of classes $\mathcal{S}$ is *correct* if function inh exists such that conditions I_1 and I_2 are satisfied, otherwise we say that the structure of classes is *incorrect*.

Hence our problem can be stated as follow: for a given structure of classes $\mathcal{S}$ detect if it is a correct one and if it is the case then produce a function inh satisfying the conditions mentioned above. In [4] we proved that this specification of the problem (refined on the base of [2]) is consistent and complete.

- (*consistency*) For any correct structure of classes there exists a solution, i.e. a function inh satisfying both conditions I_1 and I_2. Moreover, we gave an algorithm which decides whether a given structure of classes is correct and for a correct class structure constructs a function inh of desired properties I_1 and I_2.
- (*completeness*) The specification has the uniqueness property, i.e.any two solutions are equal. This is an indispensable language semantics requirement.

In the earlier paper [4] the reader will find a non-deterministic algorithm and the analysis of its correctness.

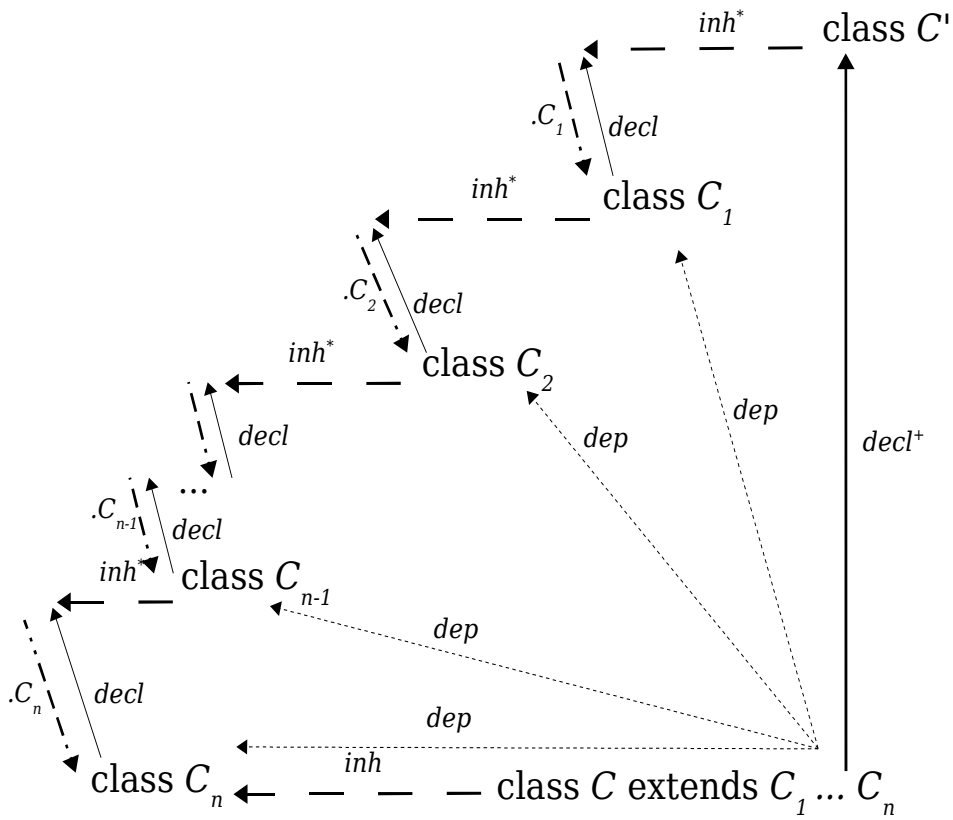

Figure 1. Downward skew inheritance.

Let K denote the class of the following header **class** C **extends** $C_1.C_2.\ ...\ .C_{n-1}.C_n$. The diagram can be viewed from several angles.

First, if we delete the arrows dep and $decl$ and keep the arrow $decl^+$, then the diagram obtained in this way commutes.

Second, the commutativity of the modified diagram illustrates the condition I_1, i.e. $inh(K) = bind(ext(K)\ in\ decl(K))$.

Third, the diagram may help to understand how to calculate the inh-arrow for the class K. In this case we assume that all other *inh*-arrows appearing in the diagram were calculated earlier. We are to identify the class M_1 of the name C_1, $M_1 = \mu\langle j, i\rangle(inh^i decl^j(K)).C_1$ is defined), then the class $M_2 = \mu i(inh^i(M_1).C_2$ is defined) of the name C_2, ... the class $M_n = \mu i(inh^i(M_{n-1}.C_n$ is defined) of the name C_n, in this order. Now we can put arrow *inh* leading from K to M_n.

Finally, we put arrows *dep*. The diagram of the structure of an entire program is not allowed to contain a cycle of *dep* arrows, c.f. condition I_2

.

3. A deterministic algorithm

This section presents a deterministic algorithm which elaborates direct superclasses. The algorithm consists of a few procedures

```
Algorithm
    DCStack := EmptyStack
    Mark Root and Object White and all other classes mark Black
    let inh(K)=null for every K ∈ Classes
    call Preorder(Root)
end Algorithm

Preorder(Classes K)
    if K is Black then
        call ComputeInhFor(K)
    endif
    for every L in sons(K) do
        {execution of this loop dpends on ordering of sons }
        call Preorder(L)
    endfor
end Preorder

CycleMessage(Classes K)
    writeln(”Dependence cycle:”)
    writeln(header(K))
    writeln(”depends on class named ”, Pop(DCStack), ”which is: ”)
    do
        writeln(header(Pop(DCStack)))
        if empty(DCStack) then exit endif
        writeln(” which in turn depends on class named:”Pop(DCStack), ”which is: ”)
    enddo
end CycleMessage

ComputeInhFor(Classes K)
    var Class∪{null} L i.e. value of variable L is either a Class or null
    { decl(K) is White }
    Mark K Gray
    if length(ext(K)) =0 then
        L := Object;
    else
        i:=1
        L:=Bind(ext(K[i], decl(K))
        {Invariant of the loop: L=bind(ext(K)|^i in decl(K)) }
        while ¬ (i=length(ext(K)) and L is White) do
```

```
            if L=null then
                writeln("undeclared class", ext(K)|^i, "in the header of class", K)
                inh(K):=Object; Mark K White
                return
            endif
            if L is Gray then
                Push(DCStack, L); Push(DCStack, ext(K)|^i)
                if K=L then
                    call CycleMessage(K)
                    L := Object
                    exit
                endif
                Mark K Black; Mark L Red
                return
            endif
            if L is Black then
                call ComputeInhFor(L)
                if not empty(DCStack) then
                    Push(DCStack, L) ; Push(DCStack, ext(K)|^i)
                    if K is Red then
                        call CycleMessage(K)
                        L:=Object
                        exit
                    endif
                    Mark K Black
                    return
                endif
            endif
            {Assertion1:  L=bind(ext(K)|^i in decl(K)) and L is White }
            if i=length(ext(K)) then exit endif
            i:=i+1 ; L:=BindInh(ext(K)[i], L)
        endwhile
        {Assertion2:   L=bind(ext(K) in decl(K)) and L is White or K is Red and L=Object }
    endif
    {Assertion3: L=bind(ext(K) in decl(K)) and L is White or K is Red and L=Object }
        inh(K):=L ; Mark K White
end ComputeInhFor

Bind(Id id, Classes K): Classes ∪ {null}
    var Class∪{null} L
    while K ≠ null do
        L :=BindInh(id,K)
        if L ≠ null then return L endif
        K := decl(K)
```

```
    endwhile
    return null
end Bind

BindInh(Id id, Classses K): Classes ∪ {null}
    var Class∪{null} L
    while K ≠ null do
        L:=K.id
        if L ≠ null then return L endif
        K :=inh(K)
    endwhile
    return null
end BindInh
```

This algorithm has several advantages over the non-deterministic one of [4]

- The algorithm is deterministic, all details necessary to implement it in practice are given,
- The algorithm is equipped with diagnostics and error recovery mechanism. It consists in colouring the visited classes i.e. nodes of structure of classes $\mathcal{S}$. The diagnostics recognizes two types of errors: a) undeclared class or b) cycle in dependence relation. The case of cycle in dependence relation is easy identifiable for the algorithm lists the pairs
  $\langle$ header of a class declaration, name of class on which the declared class depends on $\rangle$
- In both cases the error recovery uses the class *Object* as a target of inheritance.
  The error recovery makes possible to continue the static semantic analysis of Java programs.

4. Analysis of algorithm

We are going to prove that if the algorithm computes the function *inh* without signalling any error, then it is a correct solution of the problem.

4.1. Experiments

We did some experimenting before proceding to wording of lemmas and proving them. Two animations of experiments are offered at the URL: Here we quote a slide of detection of cycle in dep relation.

4.2. Proof of correctness

Lemma 4.1. The following observations are valid:

(i) Marking a node K White and assigning a value $\neq null$ to $inh(K)$ takes place together.

(ii) If a node is marked White then it is never marked again.

(iii) If value $\neq null$ was assigned to $inh(K)$ then it is never changed.

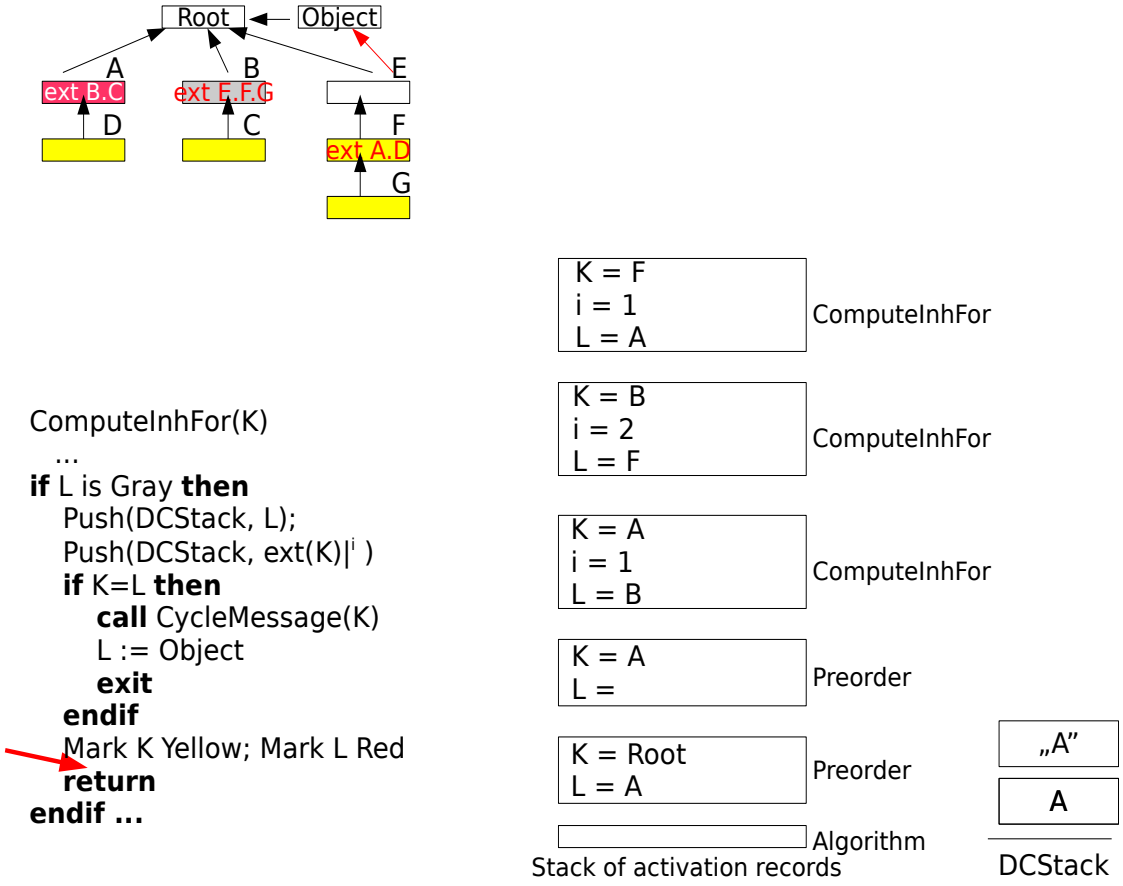

Figure 2. The cycle of dep relation of dependency is: A *dep* B *dep* F *dep* A
Explanation of the diagram:
- in left upper corner you see the structure of classes,
- in right lower corner the content of DCStack,
- to the left the stack of activation records is shown,
- in left lower corner we exhibit a part of code of currently executed method.
Relation dep is visible in activation records of ComputeInhFor.
The node K *depends* on node L.

(iv) In procedure $ComputeInhFor$ node K may get color Black. It is done if and only if instruction Push is executed.

(v) If a node is marked Grey then the stack of activation records of $ComputeInhFor$ contains a record such that its parameter K points to the node.

Proof:

(i) Look at the code of $ComputeInhFor$ and check that there only two place in the algorithm where a node K is marked White. In both cases this instruction is accompanied by the instruction $inh(K)$ takes a value. These are the only two places where such an assignment is done.

(ii) Check the code of $ComputeInhFor$.

(iii) It follows from the previous observations.

(iv) Check the code of $ComputeInhFor$.

(v) Obvious.

□

The next lemma analyzes actions taken when a node is marked Red. In order to mark a node Red the algorithm detects that the newest activation record of $ComputeInhFor$ has computed the value $L := Bind(ext(K[i], decl(K))$ and found that L is marked Grey. It means that the stack of activation records contains an element ... The algorithm pushes DCStack. Having this in mind we formulate the following

Lemma 4.2. If during an execution of the algorithm a node B is marked Red then

- activation records of $ComputeInhFor$ are closed one after another until the current activation record has the object K marked Red,
- when an activation record of $ComputeInhFor$ is going to be closed and node K is Gray then the algorithm puts two elements onto DCStack and marks K Black then the activation recod is closed, instruction return is executed,
- observe that these actions do not allow to reach *Assertion1*,
- when the current activation record of $ComputeInhFor$ has the object K marked Red then the instructions {call CycleMessage(K); L :=Object: exit} are executed causing that the algorithm skips *Assertion1* and reaches *Assertion2*.

Lemma 4.3. Whenever the algorithm reaches *Assertion1* the node L is White

Proof:
It is clear that when the algorithm reaches *Assertion1* then L can not be Grey nor Black nor null. From the above lemma we know that L is not marked Red. □

Lemma 4.4. The set $W = \{n \in Classes : \ n\, is\, White \wedge n \neq Root\}$ of nodes marked White excluding $Root$, together with the function inh is a tree. $Object$ is the root of the tree.

Proof:
Proof goes by induction w.r.t. n – the number of executed instructions "Mark K White". For $n = 0$ the tree contains only its root $Object$. Suppose that the thesis is true for a number k . At the next execution of instruction "Mark K White" we add an edge going from a non-White node K to a certain White node and we mark K White. It follows from (ii) and (iii) of Lemma 4.1 that the new graph is a tree again. □

Corollary 4.1. In every step of computation, if a node K is White then all nodes reachable by a path inh^* from K are White too.

Lemma 4.5. The following statements are invariants of every computation of the algorithm

A) If K is White then any element of the form $(decl \mid inh)^*(K)$ is White.

B) If the instruction call ComputeInhFor(K) is going to be executed then K is Black and $decl(K)$ is White.

Proof:
Proof of B) We entered *ComputeInhFor* either from *Preorder*, and then the thesis is obvious or from *ComputeInhFor*. In this case we execute the instruction call ComputeInhFor(L) and either a) $L = Bind(ext(K)[1], decl(K))$
or
b) $L = BindInh(ext(K)[i], L')$ and L' is White.
If a) then $decl(L) \in (decl \mid inh)^*(decl(K))$ therefore by inductive assumption A $dec(L)$ is White.
If b) then $decl(L) \in inh^*(L')$ therefore $decl(L)$ is White. It ends the proof of B).
Proof of A) From B) and the inductive assumption it follows that when K is marked White then $(decl \mid inh)^*(decl(K))$ is White. Using corollary 4.1 we have $inh^*(K)$ is White hence $(decl \mid inh)^*(K)$ is White which ends the proof of A). □

Lemma 4.6. When the algorithm computes $Bind$ or $BindInh$ then its second argument is a White node.

Proof:
Check the places where the instructions call Bind, respectively call BindInh, occur.Instruction call Bind occurs once before the while statement. Its second argument, $decl(K)$ is White, c.f. lemma 4.5. Instruction call BindInh occurs once, inside the while statement. Its second argument L is White c.f. lemma 4.3. □

Lemma 4.7. If each element of the form $(decl|inh)^*(K)$ is White then for every identifier id and for any later moment in the execution of the algorithm the evaluation of expression $Bind(id, K)$ (respectively, $BindInh(id, K)$ give the same result.

Lemma 4.8. In any moment of execution of the algorithm, i.e. for any function inh, and for every type name $Path$, such that $length(Path) > 0$ and for every class M

$$\begin{aligned} R = bind(Path \text{ in } M) \quad \equiv \quad & \text{L:=Bind}(Path[1], \text{M}); \\ & \textbf{for } \text{i :=2 } \textbf{to } \text{length}(Path) \textbf{ do} \\ & \quad \text{L := BindInh}(Path[\text{i}], \text{L}) \\ & \textbf{done } \{ \text{L=}R\} \end{aligned}$$

here $Path$ is conceived as an array of identifiers, $Path$[i] denotes the i-th identifier of qualified type.

Lemma 4.9. Assume that a computation of the algorithm terminates without signalling an error. The following formula is the invariant of the loop while in the $ComputeInhFor(K)$

$$L = bind(ext(K)|^i \text{ in } decl(K))$$

Lemma 4.10. Algorithm terminates.

Proof:
It suffices to show that the stack of activation records of *ComputeInhFor* is of depth limited by the number of classes. Now, recall that when an instruction call ComputeInhFor(K) is executed then node K is marked Black. Upon entrance to the procedure, node K is marked Gray. Its colour may change to White or to Black, and then the computation leaves the activation record. The colour of node K may change to Red and ComputeInhFor(K) cannot be called again. Hence it is impossible to have stack of depth bigger then n, where n is number of classes.
Observe that the number of iterations of instruction while is limited by $length(ext(K))$. □

Putting all the lemmas together we obtain the following

Theorem 4.1. If the algorithm terminates without signalling any error, then it computes function inh such that the conditions I_1 and I_2 mentioned in problem 2.1 are satisfied.

The qualitative analysis of the algorithm is completed by the following theorem

Theorem 4.2. If the algorithm terminates and signals an error then no solution exists, i.e. the structure of classes is incorrect.

Proof:
If during an execution of the algorithm a node is marked Red then a cycle in dep relation has been detected and printed out. C.f. lemma 4.2. If during an execution of the algorithm a message was printed ”undeclared class X in the header of the class Y” then the structure of classes contains the error of lackin a declaration of a class named X vusible in the place of declaration of the currently analysed class Y. For the proof see [4] □

As concerns the cost of the algorithm it can be estimated as follow: Each node is visited twice: once by the procedure ComputeInhFor and second by the procedure Preorder. During this visits the while instruction of the procedure ComputeInhFor is executed. The number of iterations is equal to the length of path appearing after extends. To this cost one must add the cost of operations Bind exectucted. In a pessimistic case the cost may be as high as $O(n^3)$. In real programs the paths occurring after the key word extends are not too long. The cost of Bind can be also less then pessimistic $O(n^2)$. In practical cases the cost of the algorithm is linear.

5. Final remarks

zmienić to!

The algorithm solves the system of two recursively defined functions. Hence, it was not obvious how to prove its correctness and completeness. The proof of correctness of non-deterministic algorithm took 10 pages.
The method of elaborating types proposed by Igarashi & Pierce [3] is highly ineffective, for it requires elaboration of each segment of a qualified type anew. The nondeterministic algorithm [4] stores the elaborated types and makes possible the further usage of earlier stored results. The deterministic algorithm uses pebbling by pebbles of differnt colours thus making the algorithm more efficient
Here comes the list of open questions

Appendix Signalling the errors

In this appendix we present a source of programs and the diagnostic produced by our algoritm

```
class A extends B.C {
    class D {}
}

class B extends E.F.G {}

class E {
    class F extends A.D {
        class G {}
    }
}
```

Below is a message of our algorithm

```
Dependence cycle :
class A extends B.C {
depends on class named B which is :
class B extends E.F.G {
which in turn depends on class named E.F which is :
```

```
class F extends A.D {
which in turn depends on class named A which is :
class A extends B.C {}
```

Observe that no existing Java compiler gives so many details on cycle in dependence relation.

References

[1] Eickel, B.: Thinking in Java, 4th edition, 2005.

[2] Gosling, J., Joy, B., Steele, G., Bracha, G.: *The Java Language Specification, 3rd edition*, Addison-Wesley, 2005.

[3] Igarashi, A., Pierce, B.: On inner classes, *Information and Computation*, **177**, 2002, 56–89.

[4] Langmaack, H., Salwicki, A., Warpechowski, M.: On an algorithm determining direct superclasses in Java-like languages with inner classes - its correctness, completeness and uniqueness of solutions., *Information and Computation*, **to appear**.

[5] Stärk, R., Schmid, J., Börger, E.: *Java and Java Virtual Machine Definition, Validation, Specification*, Springer Verlag, Berlin, 2001.