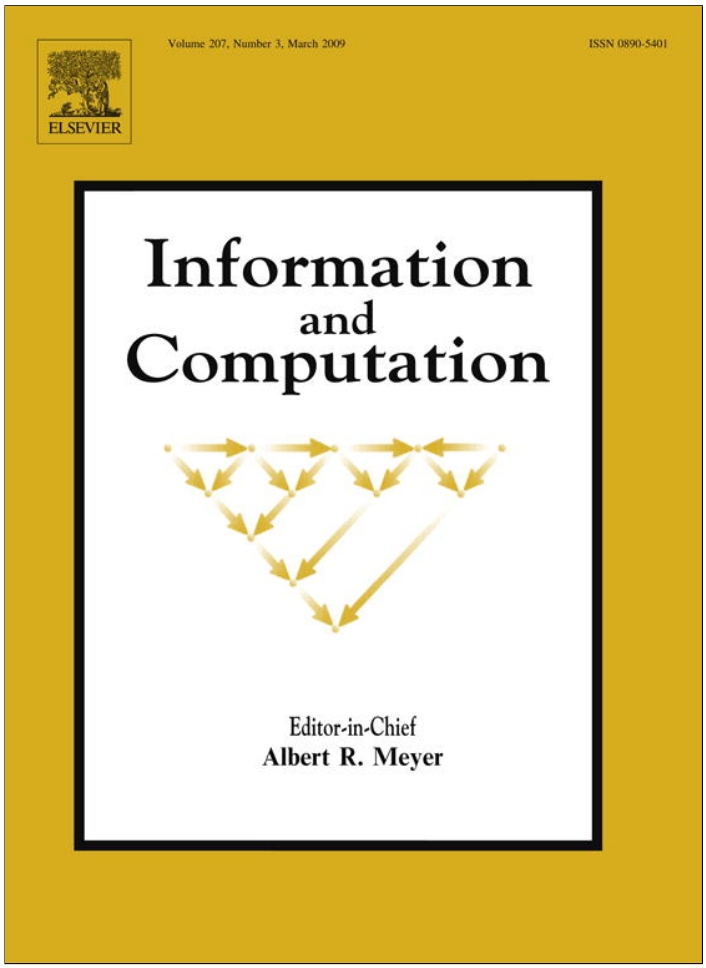

Information and Computation 207 (2009) 389–410

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/ic

On an algorithm determining direct superclasses in Java and similar languages with inner classes—Its correctness, completeness and uniqueness of solutions

Hans Langmaack [a], Andrzej Salwicki [b,*], Marek Warpechowski [c]

[a] *Institut für Informatik, Christian-Albrechts-Universität zu Kiel, Christian-Albrecht-Platz 4, D-24098 Kiel, Germany*
[b] *National Institute of Telecommunication, Szachowa 1, 04-894 Warszawa, Poland*
[c] *Institute of Informatics, Warsaw University, Banacha 2, 02-092 Warszawa, Poland*

ARTICLE INFO

Article history:
Received 6 December 2005
Revised 24 November 2008
Available online 25 December 2008

Keywords:
Object oriented programming
Inheritance
Inner classes
Direct superclass
Static semantic analysis
Static binding

ABSTRACT

Some object oriented programming languages allow inner classes. All of them admit inheritance. This combination of inner classes and inheritance is very fruitful however less known. On the other hand it creates a serious problem: how to determine the direct superclass of a given class C, i.e. the class which class C directly inherits from. For there may be several classes of the same name in one program. A specification of the problem and a non-deterministic algorithm are provided. We prove that the algorithm is *correct* w.r.t. the specification and *complete*, i.e. if the algorithm signals an error then no solution exists. We show that the specification itself has at most one solution, in other words, it is a *complete* specification. This proves also that the corresponding parts of Java Language Specification are consistent and define uniquely a fragment of Java semantics.

1. Introduction

In the paper, we address the problem of determining direct superclasses. This problem becomes hard to answer when an object-oriented programming language admits inner classes. Inner classes of SIMULA67 [13], LOGLAN'82 [2,5], BETA [3] and Java (Java 1.2 and subsequent versions) [10–12] introduce block structure for class declarations. Consider an arbitrary Java-program. The set of classes of the program together with the relation *class A is an inner class of class B* is a forest structure. (The roots of trees are top level classes.) Therefore several classes of a program may be given the same name (see Examples 1 and 2). Classes use the clause **extends** C, (to be read as "*this class inherits from a class named* C"). However, the meaning of the name C is not unique. Which of possibly many classes named C is the direct superclass of our class? The consequences of a possible error in inheriting may be dramatic if the author(s) and readers of a program interpret the meaning of inherited classes differently. Obviously, compilers are readers of programs. Therefore we postulate as a matter of course: for every program *P*, its author and the compilers should identify the direct superclasses in *the same way*. This implies the necessity of a clear and compact criterion which would guarantee the existence of a solution of the problem of determining direct superclasses, whether the solution was guessed by a programmer in an intuitive way or whether it was computed mechanically by a compiler. To find a solution is so challenging because the Java Language Specification JLS [11] is defining inheritance or superclassing rather implicitly: (1) Inheritance is defined by the help of the binding function which binds applied identifier

* Corresponding author. Fax: +48 22 512 84 00.
E-mail address: salwicki@mimuw.edu.pl (A. Salwicki).

0890-5401/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2008.12.002

occurrences in a program to their declaring occurrences. (2) The binding function on the other hand is defined by use of the inheritance. Therefore we need a more formalized specification of the problem and a constructive way of solving it, i.e. an algorithm. The algorithm should be applied as the first step in the static semantic analysis performed by the compiler.

Example 1. In the program below there are three classes named B. We can identify them as follows: class B, class A$B, class A$D$B.

```
class B { }
class A {
    class B { }
    class C extends B { }
    class D {
        class B { }
    }
    class E extends D.B { }
} //end A
```

Which class $inh(C)$ is the direct superclass of the class C? Which class $inh(E)$ is inherited by the class E?
The answer is easy, $inh(C) = A\$B$, $inh(E) = A\$D\$B$. We guessed the first answer following the usual method of static binding that for any applicative occurrence of an identifier finds a declaration of this identifier that is appropriate. The second answer was found in two steps: first we searched a declarative occurrence of D which is $A\$D$, next, we searched a class named B declared inside the class D.

The second example shows that the complexity of the problem is non-trivial.

Example 2. Consider the following classes.

```
class A extends B {  // this can be class B or A$B
    class C extends D {  // this can be class B$D or E$D
        class F extends G {} // this can be class A$E$G or E$G
    } // end C
    class E {
        class G {}
    } // end E
    class B extends E {}  // this can be class E or A$E
} // end A
class B {
    class D extends E {}  // this can be class E or A$E
} // end B
class E {
    class G extends B {}  // this can be class B or A$B
    class D {}
} // end E
```

In this example the function inh that for every class K associates with it its direct superclass $inh(K)$ may be defined on 2^6 possible ways. For we have six clauses **extends** and for each clause there are two classes of the name mentioned in it.

This example shows that searching all possible candidates for inh function is not a good approach as in general the number of candidates for the mapping inh may be exponential function of the length of a given Java program. The compiler would be stuck for indefinite amount of time.
The next example shows that the solutions may be counter intuitive.

Example 3. This example shows that the requirement "a class may not depend on itself" is essential. The natural, however complicated, requirement that all types mentioned in the extends phrases have some meaning, is not sufficient. Adding a

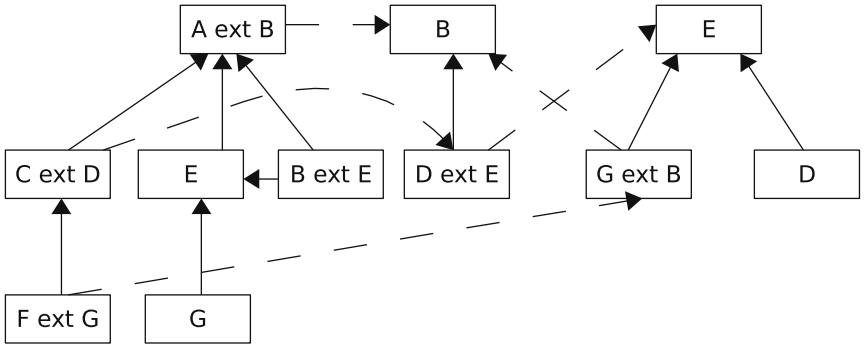

Fig. 1. The diagram of Example 2 shows the structure of classes, solid arrows lead from a class to its enclosing class. Dashed arrows lead from a class to its direct superclass, they show the unique solution *inh*.

natural, additional requirement that there is no cycle in the inheritance relation does not help. We show that there exist two different and astonishing functions, candidates for *inh*.

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

Two candidate functions: inh_1 and inh_2 may be guessed

	A	B	GDC	ICD
inh_1	GDC	G$D	G	I
inh_2	I$C	ICD	G	I

Both functions seem to be correct since $inh_i(K)$ is reestablished by binding the extends type of every class declaration occurrence K. Both functions inh_1 and inh_2 satisfy conditions mentioned in JLS [11] in 6.5.5 (later in Problem 7 formalized as condition I_1). Still we feel uneasy because the program would have two different dynamic semantics what is not appropriate for a well-formed program. Which of two is the right inheritance function? Or are both of them wrong with respect to Java's static semantics? The answer, namely the rejection of well-formedness of program Example 3 due to JLS [11], will be given later by Theorems 17, 27 and 34. Example 3 uncovers the decisive rôle of the dependency relation which JLS [11] is introducing. This example violates the requirement "no class depends on itself" of JLS [11] 8.1.4 (later in Problem 7 formalized as condition I_2). This implicit condition can not be directly included into a compiler. However, we need a precise criterion, in form of an algorithm, to be used by the constructors of compilers for some code of any length.

The fourth example demonstrates that compilers may compile the same program in different ways. It has little to do with the problem of determining direct superclasses, yet, it shows the scale of disagreement on the meaning of program.

Example 4. We asked several Java programmers to tell what the program would print. The answers came in two classes: "prints 1", "prints 2". Next, we tested the program by five Java compilers {javac, gcj, jikes, kopi, ecj }. The results were in three (!) classes. For one compiler said: The program has an error. In this way, an open question arises: *Is Java an unambiguous programming language?*

```
class B1 {
    int f() { return 1; }
    class A { int x = f(); }
    class B2 {
        int f() {return 2;}
        class A { int x = f(); }
        class B extends B1.A {}
    }
}
class UnHidden {
    public static void main( String[] argv) {
        System.out.println(new B1().new B2().new B().x);
    }
}
```

These examples show that the problem of determining direct superclasses is of importance for compiler writers and **for programmers** as well. Even short programs may create problems in proper determination of their meaning. It seems that the majority of programmers is unaware of subtle problems that can appear during their work with programs. The help provided by the reference book Java Language Specification [11] is clumsy. The book requires that a programmer reads several sections (e.g. *8.1.4 Superclasses and Subclasses, 8.5 Member Type Declarations, 6.5.5 Meaning of Type Name*.) before he/she is able to understand what is happening in a given program.
Someone may say: "your examples are unrealistic. The programmers do not write such weird programs". Let us remark that the programs become longer and more complicated than our examples. Compilers must be prepared to detect any possible error in any source code.

Another doubt may appear: "are inner classes needed at all? Some descendants of SIMULA67 such as SMALLTALK, C++, C# forbid inner classes."

It turns out that the combination of inheritance and inner classes offers many interesting possibilities:

- it allows to obtain most of the effects of multiple inheritance c.f. [19, Chapter 10],
- instead of passing classes as parameters one can extend abstract inner classes which serve as counterparts of formal parameters [20 p. 176],
- provides a convenient way to express call back objects [19],
- allows to inherit certain patterns of architecture, e.g. a class pattern of the model-view-controller system can be defined and extended by inheriting classes [2,19],
- allows to inherit protocols [2 p. 112–113],
- enables inheritance of a class put earlier into a tree-like library of classes,
- and many others.[1]

The structure of the paper is as follows: In Section 2 we introduce the necessary notations and definitions and we give the specification of the problem. Section 3 contains the algorithm. In Section 4 we analyze the correctness and the completeness of the algorithm. Section 5 is devoted to the analysis of the problem. We ask whether the specification of the problem is consistent and complete, moreover we estimate the complexity of the problem. Section 6 compares our results with earlier works and makes some final conclusions.

2. Notations and formulation of the problem

We assume that the reader is accustomed to the programming language Java [11] and the notions of class, top-level class, direct superclass.
In this section, we generalize the intuitions which arise from the examples. We begin with the observation that instead of Java-programs it suffices to consider their structures of classes. Let P be a given Java-program. We add two predefined classes {*Root*, *Object*}. We may assume that P's top level classes are contained in the additional class *Root*. We assume also that the class *Root* contains the additonal class *Object*. Now, we strip the program[2] and leave only the clauses

class A {, or
class A **extends** B {, or
(*) **class** A **extends** $B_1.B_2.\ldots.B_n${

and the corresponding closing braces }.

[1] The present authors study these possibilities for over 30 years c.f. [2,8,9] and still do not know all advantages of combined usage of inner classes and inheritance.
[2] That is we throw away all instructions and all declarations other than class declarations.

In this way, one obtains a Java-program which exhibits the structure of all classes. It contains all classes declared in the program and two predefined classes *Root* and *Object*. Each user declared class has its *name*—an identifier introduced by the declaration. Additionally we assume that the name of class *Object* is the identifier Object. Let *Classes* be the set of user defined class declarations occurrences in a program *P*. The structure of classes is equipped with a partial function *decl* which for every class *K* but *Root* points to that class in which class *K* is declared, i.e. the textually directly enclosing class. It is easy to observe that the structure $\langle Classes \cup \{Root, Object\}, decl\rangle$ of classes is a tree. Class *Root* is the root of this tree.[3] The clauses **extends** bring another function defined on the set of user defined classes, namely, for each class except *Root* and *Object* we have a type associated to it. For some classes the type is empty (in these cases the keyword **extends** is omitted), for some other classes the type consists of one class identifier, for other classes the type is a qualified type, i.e. a finite sequence of class identifiers separated by dots. If a type consists of just one identifier then it is the name of the direct superclass (the directly inherited class). One program may contain many classes of the same name which makes the problem of determining which class is direct superclass of a given class a non-trivial task. Every well-formed program satisfies the *local distinctness property* which says that every two different directly inner classes directly declared in a class have different names. The property will be useful in our considerations. Now, Java allows the types of length> 1, c.f.$^{(*)}$. The identifiers occurring in a type are names of classes. The declaring occurrence of class named B_1 should be visible from the place where the class *A* is declared and for every $1 \leq i < n$ the class B_{i+1} is a member (an attribute)[4] of the class named B_i (i.e. an inner class of B_i or an inner class of a direct or indirect superclass of class B_i).

Let *P* be a (stripped) Java program. Sometimes we shall use a formal description of the structure of classes of the program *P*:

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

where

- *Classes* is the set of classes declared (more distinctly: class declaration occurrences) in the stripped program *P*,
- *Id* is the set of identifiers found in the stripped program *P* plus the identifier Object,
- *Types* is the set of types found in the stripped program *P* after the keyword **extends**,
- $decl : Classes \cup \{Object\} \longrightarrow Classes \cup \{Root\}$
  is the function which for each class $K \in Classes \cup \{Object\}$ returns the textually directly enclosing class of the class *K*,
- $name : Classes \cup \{Object\} \longrightarrow Id$
  is the function that returns the identifier of a given class. The additional class *Root* has no name. For the class *Object* $name(Object)$ = Object,
- $ext : Classes \longrightarrow Types$ is the function which for each class $K \in Classes$ returns the (extension) type found in its extension clause. If the extension clause is omitted in the declaration of class *K* then $ext(K) = \varepsilon$.

Below we list properties of the structure $\mathcal{S}_P$.

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

Let *C* be an identifier. In the remainder of this paper we shall use partial function

$$.C : Classes \cup \{Root\} \longrightarrow Classes.$$

Let *K* be a class. The expression *K*.*C* denotes a class *Y* which is declared within class *K* and its name is *C*, *K*.*C* is defined $\Longleftrightarrow (\exists Y)(decl(Y) = K \wedge C = name(Y))$. The well-definedness of .*C* follows from the third property listed above.
One can conceive this structure as a graph. The set *Classes* is the set of nodes of the graph. Each node has two attributes associated with it: *name*—the name of the class, *ext*—the type designating its direct superclass. The function *decl* defines the edges of the graph. An example of the graph is shown in Fig. 1.
The same graph may be obtained from the SymbolTable—the data structure built by the compiler for the program *P*. One takes the SymbolTable and throws away (ignores) all irrelevant information about declarations of variables, methods, constructors, etc., only information about classes is retained.
Speaking informally, for a given structure $\mathcal{S}$ the problem is to obtain a partial function *inh*, (or equivalently, a set of edges of colour *inh*), which for every given class $K \in Classes$ 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 (static semantically) correct one. Fig. 1 has continuous edges—edges showing the *decl* function and dashed edges—edges showing *inh* function.

[3] There is a bijection between the set of *Classes* and the set *T* of finite sequences of names of classes such that a sequence $s \in T$ iff it is a code of a path leading from a top-level class to a given class. This concept is present already in [16]. For example, A.B.F denotes the class F declared inside the class B which is declared in a top level class A.

[4] "Attribute" is the jargon of SIMULA67, LOGLAN'82 and BETA, "member" is the jargon of SMALLTALK, C++ and Java.

In the sequel we shall use the following notations. Let f be a partially defined mapping $f : X \to X$. An i-th iteration of the function f is defined by induction

$$f^0(x) = x \qquad f^{i+1}(x) = f(f^i(x)).$$

The notation f^* denotes zero or more iterations of the function f, while f^+ denotes one or more iterations of the function f. Let Y be a subset of the set X. Notation $f|_Y$ denotes the restriction of the function f to the set Y. Before we specify the problem, we need definitions of a partial function *bind*, which is based on a given function *inh*, partially defined on a subset of *Classes*. Below we shall give an inductive definition of the partial function

$$bind : Types \times Classes \to Classes$$

which to a given pair $\langle type\ T, class\ C\rangle$ associates a *class D*. An equation $bind(T\ in\ C) = D$ reads informally as: the meaning of type T inside the class C is the class D, or in other words inside the class C the type T is bound to the class D. Note that the same type T may have a different meaning inside another class C'.

Definition 5 (A_1) (*base of induction 1*)**.** For any class K the meaning of the empty type ε is bound to Object. We define

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

(A_2) (*base of induction 2*). 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

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

where the pair (j, i), $j \geq 0$, $i \geq 0$, is the least pair in the lexicographic order such that the class $(inh^i decl^j(K)).C$ is defined. 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 undefined in the remaining cases.

(B) (*inductive step*). 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.

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

where $i \geq 0$, is the least natural number such that the class $(inh^i(bind(X \text{ in } K)).C$ is defined, the value of $bind(X.C\ in\ K)$ is undefined in the remaining cases.

Compare our definition with the text of JLS [11] (Sections 6.3, 6.5.5 and 8.1.4). 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. Notice that the partial function *bind* can be given an algorithmic definition (in the form of a procedure) as well, c.f. Appendix C.

The following relation *dep* plays an important rôle in the further considerations. In the description of the Java [11] it is called the dependency relation (induced by a superclassing function *inh*).
Let $ext(K)$ be the following type $C_1.C_2.\ \ldots\ .C_i.\ \ldots\ .C_n$. Then $ext(K)|^i$ denotes the initial segment $C_1.C_2.\ \ldots\ .C_i$ of length i, of the type $ext(K)$.

Definition 6. The dependency relation *dep* is

$$\begin{aligned} dep \stackrel{df}{=} \{\langle K, bind(ext(K)|^i\ in\ decl(K))\rangle :\ & \\ K \in Classes,\ & \\ 0 < i \leq length(ext(K))\ for\ ext(K) \neq \varepsilon,\ & \\ i = 0\ for\ ext(K) = \varepsilon\}. & \end{aligned}$$

The above definition can be read as follows: let a class K be of the form: **class** C **extends** $C_1.C_2.\ \ldots\ .C_i.\ \ldots\ .C_n\ \{\ \ldots\ \}$ then the class K depends on every class designated by the type $ext(K)|^i$. In JLS [11] (Section 8.1.4) one finds the following sentence: *It*

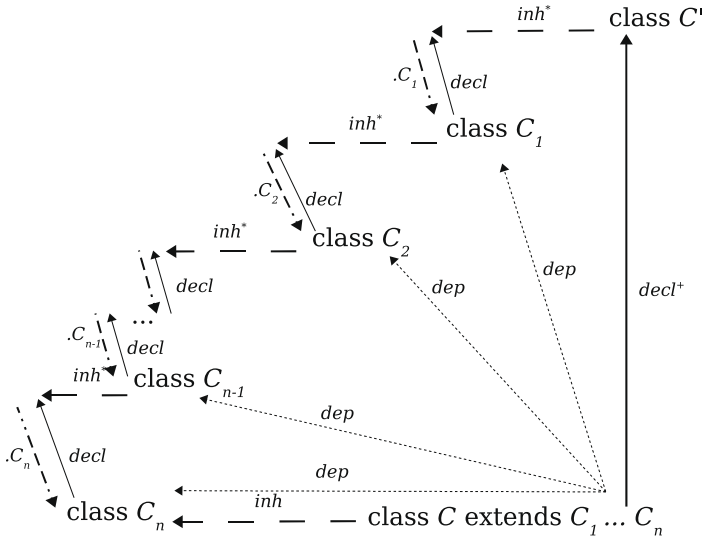

Fig. 2. Direct superclass of **class** C **extends** $C_1.C_2.\ \ldots\ .C_{n-1}.C_n$. Let K denote the class named C. The diagram can be viewed from several angles. First, if we delete the arrows *dep* and *decl* and keep the arrow $decl^+$, then the diagram obtained in this way commutes. This is so for every class in a well-formed structure of classes. Note however, that the diagrams themselves may differ accordingly to the length of type mentioned in the **extends** clause. 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 class K (note, this is not an algorithm!). In this case we assume that all other *inh*-arrows appearing in the diagram were calculated earlier. We are to identify class M_1 of name C_1, $M_1 = (inh^i decl^j(K)).C_1$ where the pair $\langle j, i\rangle$ is the least pair such that the value of the expression $inh^i(decl^j(K)).C_1$ defined, next the class M_2 of name C_2, $M_2 = inh^i(M_1).C_2$ where i is the least integer, such that the value of $inh^i(M_1).C_2$ is defined, ... class M_n of name C_n, $M_n = inh^i(M_{n-1}).C_n$ where i is the least integer such that the value of $inh^i(M_{n-1}).C_n$ is defined, in this order. Now we can put an arrow *inh* leading from K to M_n. During the above process, we put arrows *dep* leading from the class K to the classes M_i, $1 \le i \le n$. The diagram of the structure of classes enriched with *inh*-arrows and *dep*-arrows may not contain a cycle of *dep*-arrows.

is a compile-time error if a class depends on itself. The word depends in this sentence is to be meant as the transitive closure of the relation *dep*.

Fig. 2 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 7. For a given structure of classes $\mathcal{S}$, find a solution which is either a total superclassing (inheritance) function

$$inh : Classes \longrightarrow Classes \cup \{Object\}$$

or an object *ERROR* of type *Exception*, such that the following properties of *inh*, respectively, of *ERROR* object hold:
Solution *inh*: *inh* and its induced binding function *bind* (Definition 5, Eq. A_1, A_2 and B) and the dependency relation *dep* satisfy the conditions I_1 and I_2,
(I_1) for every class $K \in Classes$ the value $inh(K)$ is defined and the following equality holds

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

(I_2) The relation *dep* contains no cycle.
Solution *ERROR*: There does not exist any total inheritance function *inh* with the above mentioned properties.
The structure $\mathcal{S}$ is said *well-formed* if solution *inh* exist, otherwise it is said to be *erroneous*.

3. The algorithm

Below, we present a non-deterministic, abstract algorithm named LSWA, which computes the function *inh*. The algorithm uses function *bind*.
Data Structure: The class structure $\mathcal{S}$ of a program.
Argument: The graph G representing the structure $\mathcal{S}$.
Result: The function *inh* which for each class $A \in Classes$ shows a class B, the direct superclass of class A, or executes a command *Error* signalling that the structure of classes is erroneous.
Specification: See the Problem 7. The function *bind* satisfies the conditions mentioned in the Definition 5.

Algorithm LSWA:

Visited := {*Root*, *Object*};
inh := ∅ ;
while *Visited* ≠ (*Classes* ∪ {*Root*, *Object*})
do
  Candidates := {*K* : *decl*(*K*) ∈ *Visited* ∧ *K* ∉ *Visited*}
  if (∃*K* ∈ *Candidates*) *bind*(*ext*(*K*) in *decl*(*K*)) ∈ *Visited*
  then
    let *K* be a *Candidate* found in the above test ;
    M := *bind*(*ext*(*K*) *in decl*(*K*));
    inh := *inh* ∪ {⟨*K*, *M*⟩};
    Visited := *Visited* ∪ {*K*}
  else
    Error
  endif
endwhile

The values of function *bind* are computed using the current diagram of function *inh* computed so far. For an algorithm of bind consult Appendix C.

The word *Error* is an abbreviation of the following instruction
if $\exists K_{K\in\ Candidates}\exists i_{1\le i\le length(ext(K))}\forall j_{1\le j<i}(bind(ext(K)|^{j}\ in\ decl(K)) \in Visited$
  $\wedge\ bind(ext(K)\ |^{i}\ in\ decl(K))$ is undefined)
then
  throw new Signal_ConditionI_1_violated() // a direct superclass of *K* cannot
                                  // be detected. There is no declaration
                                  // of a class named *ext*(*K*)[*i*]
else $//\forall K_{K\in\ Candidates}\exists i_{1\le i\le length(ext(K))}\forall j_{1\le j<i}(bind(ext(K)|^{j}\ in\ decl(K))$
    // ∈ *Visited* ∧ $bind(ext(K)\ |^{i}\ in\ decl(K)) \in Candidates$)
  throw new Signal_ConditionI_2_violated() // there is a cycle in the *dep*-relation
endif

We shall prove: In this way the algorithm brings the correct diagnosis of the reasons why either it is impossible to find a function *inh* satisfying condition I_1 or if this were possible then the function's *inh* dependency relation *dep* would have a cycle.

4. Analysis of the algorithm

4.1. Correctness and completeness

We are going to prove that the algorithm *terminates* and is *correct*, i.e. that if it halts in a successful way (i.e. without signalling an error) then the computed function *inh* is satisfying the conditions I_1 and I_2. Moreover the algorithm is *complete* meaning that if it signals an error that this diagnosis is correct, i.e. there is no function *inh* satisfying both I_1 and I_2. Finally, we are going to prove the *uniqueness*. We show that any solution satisfying the conditions I_1 and I_2 is identical to the solution computed by the algorithm.

Lemma 8 (*on termination*)**.** *The algorithm terminates.*

Proof. In each step of iteration of the algorithm either the set *Visited* is increased or the algorithm signals an error and stops. It is easy to observe that the number of iterations is not bigger than the number of declared classes. □

Definition 9. A *state S* is the pair $\langle Visited_S, inh_S\rangle$ of values of corresponding variables, computed by the algorithm at the moment of testing the condition of the while instruction.

Obviously, inh_S is a set of pairs of classes, $Visited_S$ denotes a subset of the set *Classes* ∪ {*Root*, *Object*}.

Remark 10. *For every state S, the graph*

$$G_{1S} = \langle Visited_S, decl|_{Visited_S}\rangle$$

is a tree with the root Root. Graph

$$G_{2S} = \langle Visited_S \backslash \{Root\}, inh_S\rangle$$

is a tree with the root Object.

Proof. Proof goes by induction w.r.t. n, number of iterations of the algorithm. For $n = 0$ the tree G_{2S} contains only its root. Suppose that the thesis is true for a number k of iterations. In the next iteration one adds an edge going from outside the set *Visited* to a certain node in this set. Therefore the new graph is a tree again. □

Lemma 11. *For every state* $S = \langle Visited_S, inh_S\rangle$, *for every class* $K \in Visited_S$ *and for every type* P :
If $bind_{inh_S}(P \text{ in } K) \in Visited_S$ *then* $bind_{inh_S}(P|^i \text{ in } K) \in Visited_S$ *for* $1 \le i < length(P)$.

Proof. Assume the thesis of the lemma is wrong. Then there is a greatest i_0 such that $1 \le i_0 < length(P)$ and class $C_{i_0} = bind_{inh_S}(P|_0^i \text{ in } K) \notin Visited_S$. The subsequent class $C_{i_0+1} = bind_{inh_S}(P|^{i_0+1} \text{ in K}) \in Visited_S$, and, from the definition of bind, we have: $C_{i_0+1} = inh_S^k(C_{i_0}).name(C_{i_0+1})$ where k is the smallest integer such that the right side is defined. From Remark 10 we obtain that since $C_{i_0} \notin Visited_S$ then $k = 0$. Then $C_{i_0+1} = C_{i_0}.name(C_{i_0+1})$ and $decl(C_{i_0+1}) = C_{i_0}$. Again from Remark 10 since $C_{i_0+1} \in Visited_S$ then also $C_{i_0} \in Visited_S$. Contradiction. □

Lemma 12. *Let* $S = \langle Visited_S, inh_S\rangle$ *be a state.*
Let inh be an arbitrary extension of function inh_S *on the set Classes.*
(A) For every class $K \in Visited_S$, *and* $i \ge 0 : inh_S^i(K) = inh^i(K)$ *or both sides are undefined.*
(B) For every class $K \in Visited_S$ *and for every type* P : *if for every* $1 \le i < length(P)$, $bind_{inh_S}(P|^i \text{ in } K) \in Visited_S$ *then* $\forall_{M \in Classes}(bind_{inh_S}(P \text{ in } K) = M \Leftrightarrow bind_{inh}(P \text{ in } K) = M)$.

Proof. (*Proof of A*)
First we are going to prove that for every $K \in Visited_S$, $inh_S(K) = inh(K)$.
(Case 1): $K \notin \{Root, Object\}$. Then $inh_S(K)$ is defined due to Remark 10. Hence $\langle K, inh_S(K)\rangle \in inh$ since $inh_S \subseteq inh$.
(Case 2): $K \in \{Root, Object\}$. Then $inh_S(K)$ and $inh(K)$ are both undefined.
Using the remark on graph G_{2S} we conclude that $inh_S^i(K) = inh^i(K)$ or both sides are undefined.
(*Proof of B*)
(0) (*base of induction*) For types of length 0 the lemma is obvious.
(1) (*base of induction*) Consider types of length 1. Let $P = C$, C is a name of a class. We are going to prove $(bind_{inh_S}(P \text{ in } K) = M \Leftrightarrow bind_{inh}(P \text{ in } K) = M)$.
By definition $bind_{inh_S}(P \text{ in } K) = M$ iff $M = (inh_S^i(decl^j(K))).C$ where the pair $\langle j, i\rangle$ is the least pair in the lexicographic order such that the expression $(inh_S^i(decl^j(K))).C$ has a value. We are going to show that for any pair $\langle m, l\rangle$ less or equal the pair $\langle j, i\rangle$ and for any class N, $N = (inh_S^l(decl^m(K))).C \Leftrightarrow N = (inh^l(decl^m(K))).C$.
If $decl^m(K)$ has a value then it denotes a class in $Visited_S$. Put $K_0 = decl^m(K)$. Using (A) we see that for any $p : inh_S^p(K_0) = inh^p(K_0)$ or both sides are undefined.
From here one obtains $(bind_{inh_S}(P \text{ in } K) = M \Leftrightarrow bind_{inh}(P \text{ in } K) = M)$.
(I) (*induction step*) Let us assume that the lemma is true for types P of length not greater than n, $n \ge 1$. Let us consider type $P.C$. From the assumptions of this lemma we have

$$bind_{inh_S}(P.C|^i \text{ in } K) \in Visited_S \text{ for } 1 \le i \le length(P).$$

Therefore $bind_{inh_S}(P|^i \text{ in } K) \in Visited_S$ for $1 \le i < length(P)$. By inductive assumption $bind_{inh_S}(P \text{ in } K) = bind_{inh}(P \text{ in } K)$. Now we use the definition to calculate $bind_{inh}(P.C \text{ in } K)$. Arguments similar to those of point (1) lead to the result

$$bind_{inh_S}(P.C \text{ in } K) = bind_{inh}(P.C \text{ in } K)$$

or to the conclusion that both sides of the equality are undefined, which ends the proof of the lemma. □

Lemma 13. *Let* $S = \langle Visited_S, inh_S\rangle$ *be an arbitrary state. Let inh be a function satisfying condition* I_1.
Then $inh_S \subseteq inh$.

Proof. Consider the sequence of states (a computation) leading to the state S. Let us consider the longest initial segment $\{S_i\}_{i=0,\dots,q}$ of the computation such that for every state S_i the inclusion $inh_{S_i} \subseteq inh$ holds. Such segment exists for $\emptyset = inh_{S_0} \subseteq inh$. If $S_q = S$ then the thesis of the lemma is true. Suppose $S_q \neq S$. Then for a certain candidate K in the state S_q we added the pair $\langle K, bind_{inh_{S_q}}(ext(K) \text{ in } decl(K))\rangle$. Now, the function inh_{S_q}, the state S_q and the function inh satisfy the premises of the preceding lemma. We put $decl(K)$ instead of K and we put $ext(K)$ as P. From the algorithm we know that $bind_{inh_{S_q}}(ext(K) \text{ in } decl(K)) \in Visited_{S_q}$. Using Lemma 11 we have $bind_{inh_{S_q}}(ext(K)|^i \text{ in} decl(K)) \in Visited_{S_q}$ for $i = 1,\dots,length(ext(K)) - 1$. Let $M \stackrel{df}{=} bind_{inh_{S_q}}(ext(K) \text{ in } decl(K))$. Now we can apply the preceding lemma to conclude that $M = bind_{inh}(ext(K) \text{ in } decl(K))$. Since inh satisfies condition I_1 we get $\langle K, M\rangle \in inh$. Therefore the state S_{q+1} satisfies $inh_{S_{q+1}} \subseteq inh$ which contradicts our assumption. □

Remark 14. *The set $\mathbb{S}$ of states is partially ordered by the relation $\prec$ being the transitive closure of the relation of immediate successorship of states.*
Given two states S_1 and S_2 , if $S_1 \prec S_2$ then then $inh_{S_1} \subset inh_{S_2}$ and $Visited_{S_1} \subset Visited_{S_2}$.

Lemma 15. *Let S be a state $S = \langle Visited_S, inh_S\rangle$. S satisfies condition I_1 restricted in this way that in this condition the set $Visited_S$, takes place of set $Classes \cup \{Root, Object\}$, and function inh_S the place of inh.*

Proof. Let K be a class from $Visited_S \setminus \{Root, Object\}$. Let S_1 be a state earlier than S , $S_1 \prec S$, such that instruction $inh := inh \cup \{\langle K, M\rangle\}$ is going to be executed, i.e. the state S_2 next to S_1 is the first state such that $\langle K, inh_{S_2}(K)\rangle \in inh_{S_2}$. Then $inh_{S_2}(K) = bind_{inh_{S_1}}(ext(K) \text{ in } decl(K))$. By Remark 14 $inh_{S_1} \subseteq inh_{S_2} \subseteq inh_S$. From the algorithm the class $bind_{inh_{S_1}}(ext(K) \text{ in } decl(K)) \in Visited_{S_1}$. Then, by Lemma 11 for every $1 \leq i < length(ext(K))$ the class $bind_{inh_{S_1}}(ext(K)|^i \text{ in } decl(K)) \in Visited_{S_1}$. Now by the Lemma 12
$bind_{inh_S}(ext(K) \text{ in } decl(K)) = bind_{inh_{S_1}}(ext(K) \text{ in } decl(K))$
$bind_{inh_{S_1}}(ext(K) \text{ in } decl(K) = inh_{S_2}(K)$ (see above)
$inh_{S_2}(K) = inh_S(K)$ (by the above inclusion).
This ends the proof of property I_1. □

Lemma 16. *With the assumptions of the previous lemma we observe that if there exists a cycle in the relation dep_S then no one of the classes of this cycle will ever be included to the set Visited.*

Proof. Suppose that there exists a cycle in the relation dep_S.
Let $V = \{K_1 \overleftarrow{\rightarrow K_2 \rightarrow \cdots \rightarrow} K_p\}$ be this cycle. I.e. for $j = 1,\dots,p-1$ pairs $\langle K_j, K_{j+1}\rangle \in dep$ and pair $\langle K_p, K_1\rangle \in dep$. W.l.g. assume that K_1 is the class which was added to the set *Visited* as the first one. Then $K_2 = bind_{inh_S}(ext(K_1)|^i \text{ in } decl(K_1))$ for some $1 \leq i \leq length(ext(K_1))$. Let S_0, S_1 be two consecutive states such that $inh_{S_1}(K_1)$ is computed in the state S_0. By the algorithm $inh_{S_1}(K_1) = bind_{inh_{S_0}}(ext(K_1) \text{ in } decl(K_1)) \in Visited_{S_0}$. By Lemma 11, for every $1 \leq j < length(ext(K_1))$ the class $bind_{inh_{S_0}}(ext(K_1)|^j \text{ in } decl(K_1)) \in Visited_{S_0}$.
According to Lemma 12 $bind_{inh_{S_0}}(ext(K_1)|^i \text{ in } decl(K_1)) = bind_{inh_S}(ext(K_1)|^i \text{ in } decl(K_1)) = K_2$. In this way we proved that $K_2 \in Visited_{S_0}$ and $K_1 \notin Visited_{S_0}$. Contradiction! □

The above lemma leads immediately to the following

Theorem 17 (*on correctness*)**.** *Suppose that the algorithm stops without signalling an error. Then the resulting function inh satisfies the conditions I_1, I_2 and the structure of classes is well-formed.*

Now we are going to prove the completeness property of the algorithm. Namely, if the algorithm stops and signals error then no total function *inh* exists of desired properties. We begin with the remark that the instruction *Error* may be considered as an abbreviation of a conditional statement, look at Section 3. This splitting in two cases is motivated by the following observation: Should the algorithm come up with *Error* then the set *Candidates* is not empty and $ext(K) \neq \varepsilon$ for all $K \in Candidates$. For every such K there is a uniquely associated i such that $1 \leq i \leq length(ext(K))$ with $bind(ext(K)|^j \text{ in } decl(K)) \in Visited$ for all $1 \leq j < i$ and $bind(ext(K)|^i \text{ in } decl(K)) \notin Visited$. There are two possible reasons for the latter situation: Either $bind(ext(K)|^i \text{ in } decl(K))$ is undefined or a class $M \in Candidates$.
The following program examples illustrate these two cases.

Example 18. In this example no class named C is visible in the place where class A is declared which is to inherit a class named C.

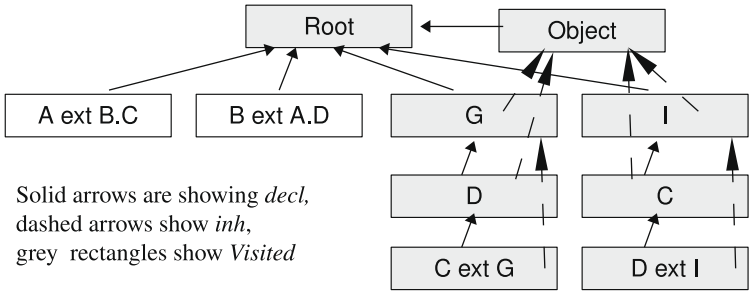

Fig. 3. Final state for Example 19 (and Example 3).

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

The algorithm terminates erroneously in final state

$$S_{fin} = \langle Visited, inh\rangle = \langle\{Root, Object, B\}, \{\langle B, Object\rangle\}\rangle.$$

Class A is the only one candidate. The value of bind(C in Root) is undefined what is showing up the first case.
A compiler should report "there is no appropriate class C declared". □

Example 19. We consider the instructive Example 3 again. It is showing up the second case. Our algorithm terminates erroneously in final state shown on Fig. 3.
A and B are the two candidates. The value bind(A in Root) is A, the value bind(B in Root) is B.

Example 20. This example has exactly one solution inh which fulfills I_1. *inh* has no cycles, but its dependency relation *dep* has a cycle.

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

Our algorithm terminates erroneously in final state
$S_{fin} = \langle\{Root, Object\}, \emptyset\rangle$.
A and B are the candidates. bind(B in Root) is B and bind(A in Root) is A, so there is a cycle $A \xrightarrow{dep} B \xrightarrow{dep} A$.
This example is so instructive also because it is showing that Igarashi's and Pierce's so called sanity conditions (6) and (7) in [16] are more liberal than the condition "*no class depends on itself*" taken from JLS [11]. Condition (6) is saying that *inh* has no cycles. Condition (7) is saying that no class A is inheriting its own inner class (no $A \xrightarrow{inh^+} B \xrightarrow{decl^+} A$ *is allowed*)."

The example 18 motivates the following

Definition 21. We say that in a given state $S = \langle Visited_S, inh_S\rangle$ the *permanent lack of a class to be inherited* occurs iff there exists a class K such that for a certain $1 \le i \le length(ext(K))$, $decl(K) \in Visited_S$ and $bind_{inh_S}(ext(K)|^i$ in $decl(K))$ is undefined and for all $1 \le j < i$ $bind_{inh_S}(ext(K)|^j$ in $decl(K)) \in Visited_S$.

Lemma 22. *If in a certain state* $S = \langle Visited_S, inh_S\rangle$ *occurs the permanent lack of a class to be inherited then no function inh exists which satisfies condition* I_1.

Proof. Suppose that a function *inh* satisfying I_1 exists. Let K and i be a class and an integer that have properties mentioned in Definition 21. By Lemma 13, $inh_S \subseteq inh$. Moreover, for all $1 \le j < i$ $bind_{inh_S}(ext(K)|^j$ in $decl(K)) \in Visited_S$. The values $inh(Root)$ and $inh(Object)$ are undefined. Observe that $decl(K) \in Visited_S$.
By Lemma 12 we have $\forall_{M \in Classes}(bind_{inh_S}(ext(K)|^i$ in $decl(K)) = M \Leftrightarrow bind_{inh}(ext(K)|^i$ in $decl(K)) = M)$.

Since *inh* satisfies I_1, $bind_{inh}(ext(K)$ in $decl(K)) = inh(K)$ is defined. Hence the righthand side of the equivalence above holds for some class M.
So $bind_{inh_S}(ext(K)|^i$ in $decl(K))$ is defined contrary to our assumption! □

Now we shall analyze the remaining case and prove that if for a certain state $S = \langle Visited_S, inh_S\rangle$ the following condition holds
(c1) the algorithm signals an error in this state, and
(c2) the property *permanent lack of a class to be inherited* does not hold for S
then there exists a cycle in the *dep* relation for every *dep* relation induced by any function *inh* satisfying I_1.

Definition 23. Each state S determines the set $Candidates_S$ which is evaluated just after the test $Visited \neq (Classes \cup \{Root, Object\})$ is performed.

We begin with an auxiliary lemma

Lemma 24. *Let S be a state such that the conditions (c1) and (c2) are satisfied then for every class $K \in Candidates_S$ there exists a class $M \in Candidates_S$ such that for a certain $1 \leq i \leq length(ext(K))$, $M = bind_{inh_S}(ext(K)|^i$ in $decl(K))$ and for all $1 \leq j < i$, $bind_{inh_S}(ext(K)|^j$ in $decl(K)) \in Visited_S$.*

Proof. Let $I_K = \{l : 1 \leq l \leq length(ext(K)) \wedge (\forall 1 \leq j < l)\ bind(ext(K)|^j$ in $decl(K)) \in Visited_S\}$. First, we show that the set I_K is non-empty. We demonstrate that $1 \in I_K$. We can assume that $length(ext(K)) \geq 1$ for in the opposite case of $length(ext(K)) = 0$ the algorithm would not signal error and add the pair $\langle K, Object\rangle$ to *inh*. Consider $l = 1$, the formula $(\forall 1 \leq j < l)\quad bind(ext(K)|^j$ in $decl(K)) \in Visited_S$ is valid. Hence the set I_K contains 1 and is non-empty.
An upper bound of the set I_K is $length(ext(K))$, hence $\max(I_K)$ is defined, denote it by i.
Suppose now, that the value of $bind_{inh_S}(ext(K)|^i$ in $decl(K))$ is undefined. It would mean that the condition of permanent lack of a class to be inherited occurs which was excluded by the assumption. Therefore there exists a class M defined by this expression $M = bind_{inh_S}(ext(K)|^i$ in $decl(K))$. We are going to show that $M \notin Visited_S$. Let us assume that $M \in Visited_S$. Suppose moreover that $i = length(ext(K))$. In this case the algorithm would add the pair $\langle K, M\rangle$ to inh_S instead of signalling error. Hence $i < length(ext(K))$. In this case $i + 1 \in I_K$ which contradicts the assumption $i = \max(I_K)$. In this way we proved that $M \notin Visited_S$.
It remains to be proved that $M \in Candidates_S$. In order to do so it suffices to show that $decl(M) \in Visited_S$. Put $C = name(M)$. Consider the case $i = 1$. We have $M = bind_{inh_S}(C$ in $decl(K))$. By the definition of $bind_{inh_S}$ $M = (inh_S^l(decl^k\ decl(K))).C$ for some l and k. Hence $decl(M) = inh_S^l(decl^k\ decl(K))$. Since $K \in Candidates_S$ we know $decl(K) \in Visited_S$. From Remark 10 we obtain $decl(M) \in Visited_S$.
Consider the case $i > 1$. From the definition of i we have
$bind_{inh_S}(ext(K)|^{i-1}$ in $decl(K)) \in Visited_S$. Let $P = ext(K)|^{i-1}$ then $ext(K)|^i = P.C$. Since $M = bind_{inh_S}(P.C$ in $decl(K))$ then by definition of $bind_{inh_S}$ we obtain

$$decl(M) = inh_S^l(bind(P \text{ in } decl(K)) \qquad forsomel \geq 0.$$

Now we apply Remark 10 and obtain $decl(M) \in Visited_S$. □

Corollary 25. *Suppose that the assumptions of Lemma 24 are satisfied. Let M be a class such that*

$$M = bind_{inh_S}(ext(K)|^i \text{ in } decl(K))$$

for a certain $1 \leq i \leq length(ext(K))$ and for all $1 \leq j < i$ $bind_{inh_S}(ext(K)|^j$ in $decl(K)) \in Visited_S$. Suppose that a function inh satisfying I_1 exists.
Then $M = bind_{inh}(ext(K)|^i$ in $decl(K))$.

Proof. By the Lemmas 12 and 13. □

Lemma 26. *If in a state S the algorithm signalled an error and there exists a function inh satisfying the condition I_1 (it implies no permanent lack of a class to be inherited due to Lemma 22) then for any natural number n one can find a sequence $K_1 \overset{dep}{\to} K_2 \overset{dep}{\to} \cdots \overset{dep}{\to} K_n$, such that all classes $K_i \in Candidates_S$, the relation $\overset{dep}{\to}$ is determined by the solution inh, according to the definition.*

Proof (*by induction*). Let $n = 1$. The set ($Classes \setminus Visited$ is non-empty). Consider a class $M \in Classes \setminus Visited_S$. Let i be the least natural number such that $decl^i(M) \in Visited_S$. Since $Root \in \{decl^j(M) : j > 0$ and $decl^j(M)$ is defined$\}$ and $Root \in Visited_S$, we know that i exists. Since $M \notin Visited_S$, $i \geq 1$. Let $K_1 = decl^{i-1}(M)$. $K_1 \in Candidates_S$.

(*inductive step*) Suppose that there exists a sequence $K_1 \overset{dep}{\to} K_2 \overset{dep}{\to} \cdots \overset{dep}{\to} K_n$ which satisfies the thesis of the lemma, i.e. $K_n \in Candidates_S$. By Lemma 24 and Corollary 25 there exists a class $M = bind_{inh}(ext(K_n)|^i \text{ in } decl(K_n))$, $M \in Candidates_S$ for a certain $1 \leq i \leq length(ext(K))$. We define $K_{n+1} = M$ and have $K_n \overset{dep}{\to} M$ which ends the proof. □

From the above considerations one obtains:

Theorem 27 (*on completeness*)**.** *If the algorithm signals Error then the structure of classes is erroneous and no function inh satisfying conditions I_1 and I_2 exists.*

Proof. Suppose that a solution *inh* exists. Since an error is signalled then either there is permanent lack of a class to be inherited (c.f. Lemma 22) and consequently *inh* does not enjoy the property I_1 or (by Lemma 26) there exists a sequence $K_1 \overset{dep}{\to} K_2 \overset{dep}{\to} \cdots \overset{dep}{\to} K_n$ of length greater than the cardinality of set *Classes*. It means that there is a cycle of *dep* arrows, it is impossible to extend the function *inh* computed so far in a way satisfying conditions I_1, I_2 and A_1, A_2, B of Definition 5. □

4.2. Fixed point theory view at algorithm LSWA

Sections 3 and 4.1 offer a constructive approach to JLS's definition of a well-formed or static semantically correct Java-program with its implicitly specified superclassing function *inh*. Every computation of LSWA is one of modularly confluent successor states approximating the uniquely determined maximal successor of the starting state, let LSWA's termination be successful (regular) or let it be erroneous. Modular confluence is a strong and far-reaching lattice theoretic property. On the other hand, LSWA's syntactical shape is that of a least fixed point approximation algorithm. So the question arises: Is there an associated cpo with a continuous functional?

Let a structure $\mathcal{S}$ of classes be given. Condition I_1 says that a function *inh* we are looking for is a fixed point of a functional on the set of all superclassing functions *inh* where each one is defined on a subset of $\mathcal{C} = Classes$ with values in $\mathcal{C}^O = \mathcal{C} \cup \{Object\}$. The so called natural functional *Bdfl* is defined by

$$Bdfl(inh)(A) \overset{df}{=} bind_{inh}(ext(A) \text{ in } decl(A))$$

which is a mapping in

$$(\mathcal{C} \overset{part}{\longrightarrow} \mathcal{C}^O) \overset{tot}{\longrightarrow} (\mathcal{C} \overset{part}{\longrightarrow} \mathcal{C}^O) .$$

$\mathcal{C} \overset{part}{\longrightarrow} \mathcal{C}^O$ is a complete partial order cpo w.r.t. set theoretical inclusion $\subseteq$ of partially defined functions.

If functional *Bdfl* were monotonous (and consequently continuous due to finiteness of $\mathcal{C} \overset{part}{\longrightarrow} \mathcal{C}^O$) then Scott's fixed point theorem [18] would yield the least fixed point

$$\mu Bdfl = \bigcup_{i \in Nat_0} Bdfl^i(inh_\perp)$$

where bottom $inh_\perp$ is the totally undefined function. In case of monotony we had here a way towards an algorithm like LSWA in Section 3. But, unfortunately, *Bdfl* is not always monotonous as the following Example 28 demonstrates.

Example 28. Showing non-monotony of the functional Bdfl.

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

We have

$$\emptyset = inh_\perp \subset Bdfl(inh_\perp) \not\subseteq Bdfl^2(inh_\perp)$$

because

$$Bdfl(inh_\perp)(B\ \$\ C) = B\ \$\ E \text{ and } Bdfl(inh_\perp)(B\ \$\ C\ \$\ D) = A$$

and

$$Bdfl^2(inh_\perp)(B\ \$\ C) = B\ \$\ E$$

and

$$Bdfl^2(inh_\perp)(B\ \$\ C\ \$\ D) = B\ \$\ E\ \$\ A \neq A.$$

So we shall modify *Bdfl*. We orient at our abstract algorithm LSWA.
We begin with the following modification (a generalization) of the notion state. Definition 9 considers states of algorithm LSWA which are states in the generalized sense too.

Definition 29. Let $inh \in \mathcal{C} \xrightarrow{part} \mathcal{C}^O$. A state is a pair $\langle dom^{RO}_{inh}, inh\rangle$ with $dom^{RO}_{inh} = dom_{inh} \cup \{Root,\ Object\}$ iff for all classes $K \in dom_{inh}$ the relations

$$inh(K) \in dom^O_{inh} = dom_{inh} \cup \{Object\},$$
$$decl(K) \in dom^R_{inh} = dom_{inh} \cup \{Root\}$$

and the equation

$$inh(K) = bind_{inh}(ext(K)\ in\ decl(K))$$

hold.

A state is uniquely represented by its associated partially defined function *inh*. Definition 29 is saying that dom^{RO}_{inh} is an initial tree of the whole *decl*-tree $\mathcal{C}^{RO} = \mathcal{C}^O \cup \{Root\}$, that K's inheritance chain $\{inh^i(K) : i = 0, 1, \cdots\}$ is remaining inside dom^{RO}_{inh} (i.e. either has a cycle or ends up in *Object* or *Root*) and that condition I_1 is satisfied, restricted to dom_{inh} as a subset of $\mathcal{C}$. *inh* and its dependency relation dep_{inh} may have cycles. In this described sense it is well allowed to call a state representing function *inh* also a state. So we shall consider the following sub-cpo

$$\mathcal{C} \xrightarrow{state} \mathcal{C}^O \quad \text{of} \quad \mathcal{C} \xrightarrow{part} \mathcal{C}^O$$

of superclassing functions which represent states. Most exciting in our deliberations on Java's superclasses are the two different states inh_1 and inh_2 in Example 3; I_1 is satisfied for both of them, they have no cycle, but their dependency relations have a cycle.

Now let us consider the activity of the body of LSWA's while loop restricted to one iteration. In case the condition of the if-statement holds (i.e. *inh* is not a maximal state) *inh* is enlarged by exactly one element, one pair $\langle K, M\rangle$. Question: Can we nominate a monotonous (and continuous) functional which is doing this enlargement? Yes, every class $A \in \mathcal{C}$ induces a so called direct successor functional

$$dsfl^A(inh) \stackrel{df}{=} inh\ \cup$$
$$\{\langle A, M\rangle : \alpha_{inh}(A)\ \wedge M = bind_{inh}(ext(A)\ in\ decl(A))\}$$

where $\alpha_{inh}(A)$ is abbreviating the formula

$$\alpha_{inh}(A) :\ decl(A) \in dom^R_{inh}\ \wedge\ A \neq Root \wedge A \neq Object$$
$$\wedge\ bind_{inh}(ext(A)\ \text{in}\ decl(A)) \in dom^O_{inh}$$

If *inh* is a state then $dsfl^A(inh)$ is well-defined, i.e. is a single-valued partial function. Namely let $A \in dom_{inh}$; then $\langle A, M\rangle$ is $\in inh$ because *inh* is a state. Let A *in* $\mathcal{C} \setminus dom_{inh}$; then in case $\alpha_{inh}(A)$ holds $M \in dom^O_{inh}$ is uniquely determined, otherwise $\langle A, M\rangle$ is not existent and, as for $A \in dom_{inh}$, $dsfl^A(inh) = inh$.

So line 9

$$M := bind_{inh}(ext(K)\ in\ decl(K));$$

in LSWA in Section 3 may be replaced by

$$M := dsfl^K(inh)(K);$$

what is preserving the semantics.

Lemma 30. *Let $A \in \mathcal{C}$ and inh be a state. Then*
(I) $inh' = dsfl^A(inh)$ is an extension of inh by at most one pair $\langle A, M\rangle$. The latter case occurs if and only if $A \in \mathcal{C} \setminus dom_{inh}$ and $\alpha_{inh}(A)$ holds, i.e. A is a candidate of inh—$A \in \mathcal{C} \setminus dom_{inh}$, $decl(A) \in dom_{inh}$—which beyond this is said to generate $\langle A, M\rangle$.
(II) inh' is a state.
(III) $dsfl^A$ is a monotonous (and continuous) functional.

Proof. (I) Clear because inh is a state.
(II) Let $K \in dom_{inh'}$. We have to show that $inh'(K) \in dom^O_{inh'}$ and $decl(K) \in dom^R_{inh'}$ and $inh'(K) = bind_{inh'}(ext(K)\ in\ decl(K))$ hold. We have two subcases
(A) $K \in dom_{inh}$ and
(B) $K = A \in dom_{inh'} \setminus dom_{inh}$.
Subcase (A) is straightforword by help of (I) and Lemmas 11 and 12.
Proof of the subcase (B): Because $inh'(A)$ is defined, $\alpha_{inh}(A)$ is holding and $inh'(A) = bind_{inh}(ext(A)\ in\ decl(A))$. Since $inh'(A) \in dom^O_{inh}$ and inh' is an extension of inh we have $inh'(A) \in dom^O_{inh'}$. Since $decl(A) \in dom^R_{inh}$ we have $decl(A) \in dom^R_{inh'}$. The last fact to prove for subcase (B) is: $inh'(A) = bind_{inh'}(ext(A)\ in\ decl(A))$. As $decl(A) \in dom^R_{inh}$, as inh' is an extension of inh and as $bind_{inh}(ext(A)\ in\ decl(A)) \in dom^O_{inh}$ we have due to Lemmas 11 and 12B)

$$bind_{inh}(ext(A)\ in\ decl(A)) = bind_{inh'}(ext(A)\ in\ decl(A))\ .$$

The left side is exactly $inh'(A)$ by definition of $dsfl^A$. Compare the proof of Lemma 15.
(III) Let $inh_1 \subseteq inh_2$ be two states and $dsfl^K(inh_1)(K) = inh'_1(K) = M$ be defined. We claim $dsfl^K(inh_2)(K) = inh'_2(K) = M$.
Case 1: $K \in dom_{inh_1}$. Then $K \in dom_{inh_2}$ and $M = inh'_1(K) = inh_1(K) = inh_2(K) = inh'_2(K)$.
Case 2: $K = A \in dom_{inh'_1} \setminus dom_{inh_1}$. Then $\alpha_{inh_1}(A)$ and $M = bind_{inh_1}(ext(A)\ in\ decl(A)) \in dom^O_{inh_1}$. Lemmas 11 and 12B are ensuring

$$bind_{inh_1}(ext(A)\ in\ decl(A)) = bind_{inh_2}(ext(A)\ in\ decl(A)).$$

So $M \in dom^O_{inh_2}$. Furtheron, due to $\alpha_{inh_1}(A)$: $A \neq Root$, $A \neq Object, decl(A) \in dom^R_{inh_1} \subseteq dom^R_{inh_2}$. So $\alpha_{inh_2}(A)$ is holding and $M = inh'_2(A)$.

Remark on the proofs of (II) and (III): The Lemmas 11 and 12 from Section 4.1 play a crucial role. The proofs of Lemmas 11 and 12 have been done for LSWA-states. But the proofs work for generalized states as well, word for word, where only LSWA-states are to be replaced by generalized states. □

We are not yet fully satisfied with our first replacement of LSWA's line 9 because a whole family of monotonous functionals $dsfl^A$, $A \in \mathcal{C}$, is involved. We want to see one single monotonous functional. Such a functional is

$$Bdfl' \stackrel{df}{=} \bigcup_{A \in \mathcal{C}} dsfl^A$$

where a union of functionals is defined in standard manner by

$$\left(\bigcup_{A \in \mathcal{C}} dsfl^A\right)(inh) \stackrel{df}{=} \bigcup_{A \in \mathcal{C}} (dsfl^A(inh)).$$

We easily calculate

$$\begin{aligned} Bdfl'(inh) &= inh \cup \{\langle A, M\rangle : A \in \mathcal{C} \wedge M = dsfl^A(inh)(A)\} \\ &= inh \cup \{\langle A, M\rangle : A \in \mathcal{C} \wedge \alpha_{inh}(A) \wedge \\ &\quad A \in \mathcal{C} \setminus dom_{inh} \wedge M = bind_{inh}(ext(A)\ in\ decl(A))\}. \end{aligned}$$

This calculation ensures single-valuedness (well-definedness) of $Bdfl'(inh)$. Furtheron, line 9 in LSWA may be replaced a second time by

$$M := Bdfl'(inh)(K);$$

in a semantics preserving manner.

Lemma 31. *Let inh be a state. Then*
(I) $inh' = Bdfl'(inh)$ *is an extention of inh.*
(II) inh' *is a state.*
(III) $Bdfl'$ *is a monotonous (and continuous) functional.*

Proof. (I) Obvious.
(II) Is ensured by Lemma 30 (II) plus either by Lemmas 11 and 12 or by modular confluence of successor states.
(III) Transfers from Lemma 30 (III). □

Now we are ready to apply the fixed point theorem:

$$Bdfl' : (\mathcal{C} \stackrel{state}{\longrightarrow} \mathcal{C}^{\mathcal{O}}) \stackrel{tot,cont}{\longrightarrow} (\mathcal{C} \stackrel{state}{\longrightarrow} \mathcal{C}^{\mathcal{O}})$$

has exactly one least fixed point

$$\mu Bdfl' = \bigcup_{i \in Nat_0} Bdfl'^{\,i}(inh_\perp)\ .$$

A computation of LSWA which terminates either successfully (regularly) or erroneously is a sequence of direct successor states

$$inh_\perp = \widetilde{inh}_0 \prec^{ds} \widetilde{inh}_1 \prec^{ds} \cdots \prec^{ds} \widetilde{inh}_m = inh_{fin}$$

where inh_{fin} is maximal successor state of $inh_\perp$ with $m \leq \kappa = card(\mathcal{C})$. In case $m = \kappa$ we have regular termination, in case $m < \kappa$ an error is reported.

Question: Is inh_{fin} the least fixed point $\mu Bdfl'$ so that LSWA can be called a least fixed point approximation algorithm? As inh_{fin} has no generating candidates inh_{fin} is a fixed point and thus

$$\mu Bdfl' \subseteq inh_{fin} .$$

On the other hand

$$\widetilde{inh}_i \subseteq Bdfl'^i(inh_\perp)$$

holds for all $0 \leq i \leq m$ and therefore

$$inh_{fin} \subseteq Bdfl'^m(inh_\perp) \subseteq Bdfl'^\kappa(inh_\perp) = \mu Bdfl'.$$

So we have proved.

Theorem 32. *LSWA is a least fixed point $\mu Bdfl'$ approximating algorithm, let the termination be successful such that the inheritance condition I_1 is satisfied, let the termination be erroneous such that $dom_{inh_{fin}}$ is a proper subset of $\mathcal{C}$.*

So both lattice theory view and fixed point view lead to the same computations realized in algorithm LSWA. We can subsume the Theorems 17 , 27 of Section 4.1 and 32 of this Section 4.2 by

Remark 33. *Let inh_0 be a partial function*
$inh_0 : \mathcal{C} \longrightarrow \mathcal{C}^O$.
The following conditions are equivalent

(i) inh_0 is result of a successful run of the abstract algorithm LSWA,
(ii) inh_0 is satisfying the properties I_1, I_2,
(iii) inh_0 is the least fixed point of functional $Bdfl'$ and inh_0 satisfies I_1.

Formulation (iii) is especially valuable because we have got rid of the condition I_2 and of mentioning any algorithm.
The non-cycling condition I_2 of the Java Language Specification JLS [11] sounds curious. But its appearing is not that astonishing because we have been able to associate inheritance with fixed point theory to solve recursive function equations. Total definedness of functions defined by such equations is proved in practice by the help of mappings into a Noetherian order. And vice versa: Total definedness implies existence of a "natural" Noetherian order. But such natural Noetherian orders are hard to describe and to apply in a conventional way. I_2 seems to present a natural Noetherian order.[5] What is interesting: Theorem 27 is a non-conventional proof of total definedness by the help of this Noetherian order.

5. Analysis of problem

5.1. Consistency and completeness of the specification

Specification of a problem is said to be *consistent* if there exists a solution of it. Considerations of the preceding section lead to the conclusion that the Problem 7 is consistent. For every well-formed structure $\mathcal{S}$ of classes there exists a solution *inh* and a signal is raised if $\mathcal{S}$ is erroneous.
Next question we would like to address is: how many correct solutions may Problem 7 possess?

Theorem 34 (*on determinacy of algorithm and the uniqueness of solution*)**.** *If there exist two functions inh_1 and inh_2 such that both satisfy conditions I_1 and I_2 then they are equal: $inh_1 = inh_2$.*

Proof. By the completeness property (Theorem 27) and termination property (Lemma 8) if a solution exists then a computation of the algorithm stops successfully with a result *inh* . Any correct final result *inh* of the algorithm is contained in any solution (c.f. Lemma 13). Hence $inh \subseteq inh_1$ and $inh \subseteq inh_2$. Functions inh_1 and *inh* are defined on the same set *Classes*. Hence they are equal. The same holds for inh_2 and *inh*. □

[5] Sometimes it is named: *well-founded order*.

In this way we show: if Problem 7 has a solution then it is a unique one. Hence we proved that the specification of the problem is complete. We have proved Theorem 34 without fixed point theory. Note that uniqueness of the solution can also be achieved as a result of fixed point considerations, namely of statement (iii) of Remark 33 at the end of Section 4.2.

5.2. Estimation of complexity of the problem

Let the number of classes in a given Java program be n. This is less than the length of the program. It is relatively easy to give the lower bound of the problem. It is not less than $\Omega(n)$. Our non-deterministic algorithm is determinate and therefore it can be viewed as a an abstraction of a family $\mathcal{C}$ of deterministic algorithms. Consider a deterministic version of the presented algorithm. Observe that the quantifiers appearing in the algorithm can be replaced by finite disjunctions and conjunctions. They in turn can be calculated by iterative instructions. It suffices to insert a loop

---

GoodCandidates := ∅;
for $K \in$ *Candidates*
do
  if *bind(ext(K) in decl(K))* $\in$ *Visited*
  then
    GoodCandidates := *GoodCandidates* ∪ *K;*
  endif
endfor

---

and to use any algorithm to choose one element of the set *GoodCandidates*.
It turns out that, analyzing the non-deterministic algorithm, we have proved properties of the family $\mathcal{C}$ of deterministic algorithms.

Corollary 35. *Any deterministic algorithm $\mathcal{A}$ of the family $\mathcal{C}$ is correct, complete and all of them bring the same solution.*

Now we can estimate the cost of such algorithm. The main loop is repeated n times. Choosing a candidate needs n steps. Calculating the *bind* function may be estimated as $O(n^2 + n{*}l)$ where l is the maximal length of type $ext(K)$. Hence the upper bound of complexity of the problem is $O(n^3{*}max(n, l))$. This is a pessimistic estimation. We must add the following remark, a qualified type, i.e. its path expression may be of any positive length. Look at the following example.

Example 36. It shows that, at present, a path expression appearing after **extends** may have an arbitrary length and remains sound.

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

There is a substantial gap between lower and upper bounds of complexity of the problem. We believe that better algorithms of lower cost exist.

6. Final remarks and the conclusion

6.1. On related work, especially of Igarashi and Pierce

We begin this Section with the comments on SIMULA67 and LOGLAN'82 programming languages.

6.1.1. SIMULA67

In this language the type of direct superclass is designated by a single identifier. The direct superclass (or the prefixing class, in the jargon of SIMULA) has to fulfill much simpler condition

(I_{SIMULA}) for every class X, $decl(inh(X)) = inh^i(decl(X))$

where i is the least non-negative integer such that above equality is holding.
It can be proved that if $inh(X)$ is defined then $\exists k\ \ decl^k(inh(X)) = decl^k(X)$, it means that the direct superclass of a given class is either a sibling of the extended class or more generally, must be found on the same level of *decl*-tree as the class itself. We say that SIMULA67 admits the *horizontal inheritance*.
Due to this simpler requirement, the task of determining the direct superclass is easy. For example, the algorithm bind may be much shorter and simpler. The same applies as well to the algorithm of determining the direct superclass.
On the other hand the requirement that the inherited class and the inheriting class are on the same level of the structure of inner classes (also called the tree of nesting modules) makes extensions of library of classes impossible. Simula67 has only two classes in its library: SIMSET and SIMULATION. On the other hand, due to the above mentioned restriction, Simula67 can use the Display Vector mechanism [4] of Algol60 without problems.

6.1.2. LOGLAN'82

As in Simula the type denoting the direct superclass is designated by a single identifier. No restriction on the level of direct superclass is imposed. It means that the class named B must be visible from the place where the class A is declared

$$inh(A) = bind(B \text{ in } decl(A)).$$

This kind of inheritance can be described as *upward skew inheritance*, for the direct superclass is on not lower level of *decl*-tree than the class itself. The algorithm determining the direct superclasses for LOGLAN'82 is much simpler than the LSWA algorithm presented above. It can be compared to topological sort.
The library of classes may be extended at will. It is worthwhile to mention that LOGLAN'82 admits inheritance in all modules: procedures, functions, blocks, classes, coroutines and processes. In the papers [5–7] the problem of maintaning the Display Vector was addressed and solved.

6.1.3. BETA

The situation in BETA [3] is different, but no less complex than the one in Java. Inheritance in BETA is dynamic, it involves objects, not only names of classes. Note, BETA as LOGLAN'82 admits inheritance of patterns in procedures, functions, classes.

6.1.4. Java

The problems of Java inheritance have been studied among others by Igarashi and Pierce [16]. The scope of their paper is much broader, they present a formal semantics for (essentially) a subset FJI of new Java [11], Featherweight Java with Inner classes. Usual Java-programs are assigned their semantics via semantics of corresponding FJI-programs. It is characteristics of FJI that every extension clause is the complete names path of the direct superclass plus of all enclosing classes where there are not allowed identifier repetitions in a path. Due to the local distinctness property and the required visibility of top level class names there is exactly one inheritance function *inh* per program which satisfies condition I_1 of Section 2.

But not every syntactically correct FJI-program is a well-formed FJI-program, i.e. one which can be assigned an appropriate dynamic semantics. Igarashi and Pierce require so called sanity conditions to be fulfilled. Condition (6) says: *inh* has no cycles. Condition (7) says: There is no class which has any direct or indirect inner class as its direct or indirect superclass. These sanity conditions should correspond to condition I_2 in Section 2 which expresses that the dependency relation is free of cycles, see Java Language Specification [11] (Section 8.1.4, Superclasses and Subclasses).

However, the sanity conditions and the conditions I_1, I_2, restricted to FJI, are not equivalent. FJI is more liberal. Example program 20 in Section 4.1 is a drastic counter example of equivalence. Example 20 is a well-formed FJI-program, but algorithm LSWA reports an error as we have seen in Section 4.1: Condition I_2 is violated, the dependency relation has a cycle.

Igarashi and Pierce propose an Elaboration of Types calculus—let us call it IPET—which allows to infer binding. Inheriting is a special case of binding. $P \vdash X \Rightarrow T$, read "type X is elaborated to class T in class P", is what we would express $bind_0(X\ in\ P) = T$ or $bind_{inh_0}(X\ in\ P) = T$. Inferring in a general top-down manner does not work because there is one inference rule, namely ET-SimpEncl, with a metatheoretic premise $P \vdash X.\ D\ \Uparrow$ which means: "There is no derivation of $P \vdash X.\ D\ \Rightarrow T$ for any class T ".

In our opinion it is a serious methodological error to mix a theory and its metatheory. Such mixing leads to paradoxes frequently. The authors of [16] give no evidence that such a paradox will not appear.

They recommend to read the rules in a bottom-up manner and so to interpret them as a generalized program procedure, implemented and executed by help of consecutive run-time stacks of procedure activation records [4]. Generalized means: We have not only pushing-down and popping-up of activation records, but we have also backing-up in case there is some evidence that all actions whatsoever after an activation $P.C\ \vdash D$ are never resulting in any class T (see rule ET-SimpEncl). Even if such evidence is showing up, e.g. by cycling or other infinitely expanding run-time stacks, we are to know which are correct back-up states in order to guarantee determinate, non-multivalued results.

Program examples demonstrate that we need clearer correctness, completeness and termination criteria for IPET. We would like to consider calculus IPET rather a *method* than an algorithm. It is possible to repair calculus IPET and to transform

algorithm LSWA and its binding function towards a calculus without metatheoretic premises where all inferences can be done in a top-down manner, see a forthcoming article.

6.2. Conclusion

We formulated a specification of the problem of identifying direct superclasses and proved that the problem either has no solution or if a solution exists then it is the unique one. The formulation follows the pattern of static binding [6,9,15] of applied occurrences of identifiers with the proper declarations of identifiers. In our article we have extended this principle towards types, whether they are simple identifiers or qualified types. Next, we proposed an algorithm which finds a solution for a given class structure or answers that no solution exists. In fact we deal with a class of deterministic algorithms of $O(n^4)$ pessimistic cost.

The structure $\mathcal{S}$ of classes is a part of the SymbolTable data structure. The algorithm we proposed can be made more efficient if more information is used which is contained in the SymbolTable. This or a similar algorithm must be executed before further static semantic analysis can be executed by a compiler. We believe that admitting local classes (classes declared in methods) and anonymous classes (they correspond to blocks inheriting from classes in SIMULA67 and LOGLAN'82) will introduce only minor modifications to our algorithm.

The authors of new Java Language Specification JLS [11] have had a very intriguing idea how to characterize in an implicit style binding of identifiers and inheritance (superclassing) of classes in Java-programs with inner classes. The total definedness condition I_1 and especially the no cycling condition I_2 as we have formalized JLS [11]'s idea are looking, at a first glance, curious, ad hoc and hard to accomplish both by Java-compilers and by Java-programmers. This article is presenting an algorithmic access to find appropriate solutions, an access which has been fully justified by rigorous proofs of correctness, completeness and uniqueness. What is most interesting: The fixed point theoretic roots of the problem and of the algorithm have been uncovered. Strong theoretical connections assure that ideas of programming language designers and practitioners will achieve lasting importance [1]. Other authors have different views on binding and inheritance as we have found out. Their different views have been a strong incentive to find out how far their and our views are lying apart and whether a similar theory like ours might be created for other views.

The results of this paper can be viewed as follow: we proved that the definition of direct superclasses contained in JLS [11] is consistent and complete (however clumsy). We offer a family, c.f. Section 5.2, of algorithms, any algorithm of this family may be included by a Java compiler as a first step in static semantic analysis of Java programs.

Acknowledgments

The present authors wish to express their gratitude to the four anonymous reviewers for the criticism and helpful suggestions.

Appendix A. Example 4 ctd.

We saw in Example 4 that the program is differently understood by different compilers and by different people. Now we apply a refactoring transformation to the program of the example. We extract class A that appears twice.

```
abstract class AF {
    abstract int f();
    class A{ int x = f(); }
}
class B1 extends AF{
    int f(){ return 1;}
    class B2 extends AF {
        int f(){ return 2;}
        class B extends B1.A{}
    }
}
class Hidden{
    public static void main( String[] argv){
        System.out.println(new B1().new B2().new B().x);
    }
}
```

We believed that this transformation should lead to an equivalent program. In our opinion this equivalence transformation is a basic dynamic semantics property of inner classes with inheritance. We were astonished when it turned out that the

two programs are not equivalent. We tested 5 Java compilers. It turned out that one compiler signalled that the program contains an error. Four compilers translated the program and printed "2". The previous version of the program—the original Example 4—prints "1" (3 compilers) and "2" (1 compiler).

After decompilation one sees that compilers translate "**class** B **extends** B1.A{}" as "**class** B **extends** AF.A{}".
This example suggests that one should apply refactoring transformations with care. Some IDE for Java, e.g. Eclipse, offer a menu of refactoring transformations. We do not know whether the transformations are accompanied by correctness proofs. Is it possible at all?

Appendix B. On modular confluence

Lemma 37. *The transitive and irreflexive partial order $\prec$ in the set S of algorithm states is modularly confluent, i.e. if a state S_0 has two different immediate successor states S_1 and S_2 then there exists a state S_3 which is the immediate successor of both states S_1 and S_2 such that the following diagram commutes.*

$$\begin{array}{ccc} S_0 & \xrightarrow{K_1} & S_1 \\ {\scriptstyle K_2}\downarrow & & \downarrow{\scriptstyle K_2} \\ S_2 & \xrightarrow[K_1]{} & S_3 \end{array}$$

Proof. Let K_1, K_2 be two candidates which, if chosen in S_0, lead to S_1, S_2, respectively. Consequently, there exist classes $M_1,\ M_2 \in \mathit{Visited}_{S_0}$ such that

$$M_1 = \mathit{bind}_{\mathit{inh}_{S_0}}(\mathit{ext}(K_1) \text{ in } \mathit{decl}(K_1))$$

and

$$M_2 = \mathit{bind}_{\mathit{inh}_{S_0}}(\mathit{ext}(K_2) \text{ in } \mathit{decl}(K_2)).$$

Observe that $S_1 = \langle \mathit{Visited}_{S_0} \cup \{K_1\},\ \mathit{inh}_{s_0} \cup \{\langle K_1,\ M_1\rangle\}\rangle$ and $S_2 = \langle \mathit{Visited}_{S_0} \cup \{K_2\},\ \mathit{inh}_{s_0} \cup \{\langle K_2,\ M_2\rangle\}\rangle$. It is clear, that in the state S_1 the class K_2 remains a candidate (in the state S_2 the class K_1 remains a candidate). By Lemmas 12 and 11 and Remark 14 we obtain $M_1 = \mathit{bind}_{\mathit{inh}_{S_2}}(\mathit{ext}(K_1) \text{ in } \mathit{decl}(K_1))$ and $M_2 = \mathit{bind}_{\mathit{inh}_{S_1}}(\mathit{ext}(K_2) \text{ in } \mathit{decl}(K_2))$. Therefore the algorithm has further states S_2' and S_1'. We have $S_2' = \langle \mathit{Visited}_{S_0} \cup \{K_1,\ K_2\},\ \mathit{inh}_{s_0} \cup \{\langle K_1,\ M_1\rangle, \langle K_2,\ M_2\rangle\}\rangle\ = S_1'$. Hence state S_1' is the desired state S_3. □

From the Lemma follows:

Proposition 38. *Any maximal state, w.r.t. relation $\prec$ is uniquely defined.*

The following theorem extends the observations of Theorem on determinacy and uniqueness stating that the non-deterministic algorithm is determinate, not only for successful but also for erroneous termination.

Theorem 39. *The algorithm terminates with a unique final state.*

Proof. Due to the previous Propositon it is enough to show that every final state of the algorithm is also the maximal one, i.e. has no direct successor. It is obviously true for a successful termination. When an error is signalled the algorithm guarantees that no normal alternative continuation is possible. So, an erroneous final state is also a maximal one. □

Appendix C. An algorithm bind

Data structure: structure of classes $\mathcal{S}$ and partial, cyclefree function *inh*.
Arguments: type T and class K. Type T may be empty or a class identifier C_1 or a qualified type of the form $C_1.C_2.\ldots.C_n$. In the latter cases $\mathit{head}(T) = C_1,\ \mathit{tail}(T) = C_2.\ldots.C_n$.
Result: the class named C_n which is denoted by type T, visible from class K, respectively, class N named Object.
Specification: see Definition 5.

```
Algorithm:
    found := false;
    if T is empty sequence
    then
        found := true;
        N := Object
    else
        M := K; // j := 0
        C := head(T);
        while ¬ found ∧ M ≠ none
        do
            M' := M; // i := 0
            while ¬ found ∧ M' ≠ none do
                N := M'.C;
                // check if class N named C is son of class M';
                if N ≠ none
                then
                    found := true;
                    // return this class N
                else
                    M' := inh(M'); // i := i + 1
                endif
            endwhile; // either found or M' = none
            if ¬ found then M := decl(M); /* j := j + 1 */ endif
        endwhile; // found or M = none
        if ¬ found
        then
            throw new SignalClassNotDeclared()
        else // we have found class N named head(T) = C1,
         // pair⟨j, i⟩ is the least in the lexicographic order s.t. N = inh^i(decl^j(K)).C
            while not empty tail(T)
            do
                T := tail(T); C := head(T); M' := N;//k := 1; C_k := N
                found := false; // i := 0 ,
                while ¬ found ∧ M' ≠ none
                do
                    N := M'.C;
                    // check if class N named C is son of class M';
                    if N ≠ none
                    then
                        found := true; // k := k + 1; C_k := N
                    else
                        M' := inh(M'); // i := i + 1
                    endif
                endwhile; // either found or M' = none
                if ¬ found then throw new Error() endif;
            endwhile; // tail(T) is empty
        endif;
        result := N
    endif;
    // result = the class N = the meaning of type T in the class K
```

Lemma 40. *If the algorithm terminates successfully the final value of variable result is equal to bind*$(T$ **in** $K)$.

One may ask whether testing against cycles in *inh* should be added into this algorithm? Our answer is as follows: this algorithm *bind* is used by a compiler twice. First usage is after parsing and before static semantic analysis is done. In this phase, the *bind* algorithm cooperates with the algorithm LSWA of the Section 3. And it is the latter algorithm which detects cycles in the dependency relation. Second usage of algorithm *bind* takes place in static semantic analysis. Then no error of cycling in *inh* can happen since all these errors were detected earlier.

References

[1] M. Abadi, L. Cardelli, A Theory of Objects, Springer, New York, 1996.
[2] W.M. Bartol et al., The Report on the Loglan'82 Programming Language, PWN, Warszawa, 1984.
[3] O.L. Madsen, B. Moeller-Pedersen, K. Nygaard, Object Oriented Programming in the BETA Programming Language, Addison Wesley/ACM Press, 1993 see also Beta Programming Language, 2001. Available from: <http://www.daimi.au.dk/~beta/>.
[4] E.W. Dijkstra, Recursive programming, Numerische Mathematik 2 (1960) 312–318.
[5] W.M. Bartol, A. Kreczmar, A.I. Litwiniuk, H. Oktaba, Semantics and implementation of prefixing on many levels, in: A. Salwicki (Ed.), Proceedings Logics of Programs and Their Applications, LNCS, vol. 148, Springer, Berlin, 1983.
[6] M. Krause, A. Kreczmar, H. Langmaack, A. Salwicki, Specification and Implementation Problems of Programming Languages Proper for Hierarchical Data Types, Bericht No. 8410, Institut für Informatik, Christian-Albrechts-Universität, Kiel, 1984.
[7] M. Krause, A. Kreczmar, H. Langmaack, M. Warpechowski, Concatenation of program modules an algebraic approach to the semantics and implementation problems, in: A. Skowron (Ed.), Proceedings Computation Theory, LNCS, vol. 208, Springer, Berlin, 1986, pp. 134–156.
[8] G. Cioni, A. Salwicki (Eds.), Object Oriented Programming, Academic Press, London, 1987.
[9] A. Kreczmar, A. Salwicki, M. Warpechowski, Loglan'88—Report on the Programming Language, LNCS, Springer, Berlin, 1990.
[10] Inner Class Specification, 1997. Available from: <http://java.sun.com/products/jdk/1.1/docs/guide/inner/classes/>.
[11] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, third ed., Addison-Wesley, 2005. Available from: <http://java.sun.com/docs/books/jls/>.
[12] R. Stärk, J. Schmid, E. Börger, Java and the Java Virtual Machine Definition, Validation, Specification, Springer, Berlin, 2001.
[13] O.-J. Dahl, K. Nygaard, Class and subclass declarations, in: J.N. Buxton (Ed.), Simulation Programming Languages, Proc. IFIP Work. Conf. Oslo 1967, North Holland, Amsterdam, 158-174, 1968 Simula Programming Language, 2002. Available from: <http://www.iro.umontreal.ca/~126 simula/>.
[14] A. Salwicki, M. Warpechowski, Combining inheritance and nesting of classes together: advantages and problems, Proc. Conf. CS&P', Humboldt Universität, Berlin, 2002, pp. 305–316.
[15] H. Langmaack, Consistency of inheritance in object-oriented languages and of static, ALGOL-like binding, in: O. Owe, S. Krogdahl, T. Lyche (Eds.), From Object-Orientation to Formal Methods: Dedicated to The Memory of Ole-Johan Dahl, LNCS, vol. 2635, Springer, 2004, pp. 209–235.
[16] A. Igarashi, B. Pierce, On inner classes, Information and Computation 177 (2002) 56–89.
[17] B. Stroustrup, The C++ Programming Language, third ed., Addison Wesley, 1997.
[18] J. Loeckx, K. Sieber, The Foundation of Program Verification, Wiley-Teubner, 1984.
[19] B. Eickel, Thinking in Java, fourth ed., Prentice Hall, 2005.
[20] G. Mirkowska, A. Salwicki, Algorithmic Logic, PWN and D. Reidel, Warsaw & Dordrecht, 1987.