Warsaw University
Faculty of Mathematics, Informatics and Mechanics

Marek Warpechowski

On the determination of extended classes and some other problems in Java programming language

PhD dissertation

Supervisor
Andrzej Salwicki, Ph.D. Dr.,Sc.
National Institute of Telecommunications
Warszawa, POLAND

Contents

Preface v

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Four examples . . . 1
1.3 Discussion . . . 4

2 Formulation of the problem **7**
2.1 Preliminaries . . . 7
2.2 The algorithm computing function bind . . . 12

3 Non-deterministic algorithm **15**
3.1 The algorithm . . . 15
3.2 Analysis of the algorithm . . . 16
3.2.1 Correctness . . . 16
3.2.2 Completeness . . . 19

4 A deterministic algorithm **23**
4.1 The algorithm . . . 23
4.2 Analysis of algorithm . . . 26
4.2.1 Experiments . . . 26
4.2.2 Proof of correctness . . . 26
4.2.3 Complexity of the algorithm . . . 30
4.3 Remarks on efficiency and error recovery . . . 30
4.4 Signalling the errors . . . 30

5 Remarks on earlier work **33**
5.1 Introduction . . . 33
5.2 Igarashi's and Pierce's calculus IPET for elaboration of types . . . 35
5.3 Langmaack's, Salwicki's and Warpechowski's binding functions $bind_{inh}$ compared with IPET . . . 38
5.4 On another binding function $Bind_{inh_{B0}}$ which satisfies all rules of IPET . . . 40
5.5 The dilemma with IPET's rule III. (ET-SimpEncl) . . . 46
5.6 What was left open by Igarashi and Pierce . . . 47

6 Static and runtime binding **49**
6.1 Binding of variables . . . 49
6.2 Instantiation of classes . . . 50
6.2.1 Static typing of instantiation expression . . . 50
6.2.2 How to generate class instance using class instance generator . . . 51

6.2.3 How to find the proper class instance generator from the place of class instantiation. 53

7 Conclusions 55

Preface

This thesis is a part of a more general work concerning the problem of identifier binding in Java. Java is concerned here as most general of languages admitting both inheritance and inner classes. The problem of identifier binding consist in assigning the proper meaning to the occurrences of identifiers either at compile time or at run time. At the compile time we distinguish between defining and applied occurrences. The meaning of a defining occurrence of an identifier is the definition which the identifier is connected with. Each applied occurrence of an identifier must be bound to the proper defining occurrence. If we restrict our problem to the variable identifiers then at the compile time we will be interested in types of these variables. But at runtime we will be interested first of all in the location of the variable in the memory, and in most cases in the value of the variable. Since a type of the variable can be read from the declaration of this variable we can say that at compile time each variable is bound to its declaration while at runtime each variable is bound to the location in memory, which was reserved specially for this variable.

Now consider class identifiers. During compilation we are interested in binding each applied occurrence of class identifier to the declaration of the corresponding class. Then during runtime, while creating an object of a class K, we are looking for an environment which will be used for binding the nonlocal variables occurrences within methods of the class K.

Unexpectedly it turned out that the problem of binding class identifiers is especially difficult during the compilation of a program. The source of the difficulty is the fact that in Java the extended class may be denoted not only by stand alone class identifier but also by qualified type, which is a sequence of class identifiers separated by dots. We must deal here with the mutually recursive definition of binding. Binding of qualified types uses the inheritance function, while the definition of this function uses the binding of qualified type. The problem reduces to inventing the proper order of computing of the inheritance function as it will be shown in sections 3.1 and 4.1. The additional problems are checking of well-formedness of a program and the detection and correction of eventual errors.

The structure of the thesis is as follows. The first chapter *"Introduction"* exposes the problem through a series of examples. The next chapter formulates the problem of identification of direct superclass in the language of a graph of classes of a given program. Chapter 3 presents a non-deterministic algorithm and the proof of correctness. In the subsequent chapter a deterministic algorithm is presented and analyzed. Chapter 5 analyzes earlier work on the problem and in particular the work of A.Igarashi and B.Pierce. The structure of the chapter is as follows: The section 5.2 presents the calculus of Igarashi and Pierce. Examples are given showing that the calculus is inconsistent. We are giving a simple remedy to it. In the section 5.3 we translate the inference rules of the IPET calculus in such a way that the phrase "*the meaning of the type X in the environment P is the class T*" is now expressed by the formula $bind(X \text{ in } P) = T$ and we show that the function *bind* calculated by the algorithm of the sectiion 3.1 is a model of the IPET calculus. Next section 5.4 shows that there is another concept *Bind* of *binding* function and that *Bind* is modeling the IPET calculus as well. In the section 5.5 we show that the intersection of two

models needs not be a model. In this way one should abandon the hope that by adding the, metatheoretic, phrase, "*take the least model*" of the models of IPET calculus the task of identifying the direct superclass of classes of a Java program may be completed.

The last chapter exposes the application of the results obtained above during compilation and latter during the execution of programs.

Chapter 1

Introduction

1.1 Motivation

In this thesis 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 [9], Loglan'82 [1] [3], BETA [2] and Java (Java 1.2 and subsequent versions) [6], [7], [8] 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 the 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 [7] is defining inheritance or superclassing rather implicitly: 1) Inheritance is defined by the help of the binding function which binds applied identifier 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.

1.2 Four examples

Example 1.2.1. *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 1.2.2. *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 1.2.3. *This example shows that the requirement (taken from JLS[7]) "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 natural, additional requirement that there is no cycle in the inheritance relation does not*

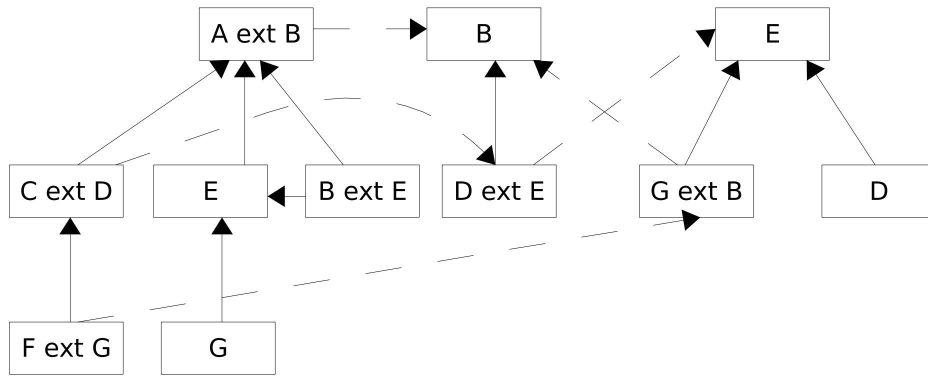
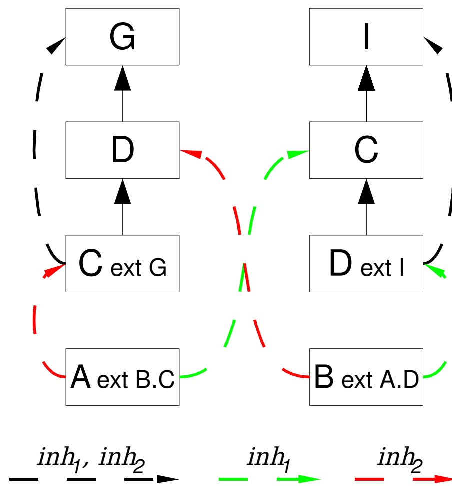


Figure 1.1: The diagram of Example 1.2.2 shows the structure of classes, solid arrows lead from a class to its enclosing class. Dashed arrows lead from a class to its direct superclass, they show the unique solution *inh*.

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 [7] in 6.5.5 (later in Problem 2.1.6 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 [7], will be given later by Theorems 3.2.10 and 3.2.20. Example 1.2.3 uncovers the decisive rôle of the dependency relation which JLS [7] is introducing. This example violates the requirement "no class depends on itself" of JLS [7] 8.1.4 (later in Problem 2.1.6 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 1.2.4. *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?

```
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 Test {
    public static void main( String[] argv) {
        System.out.println(new B1().new B2().new B().x);
    }
}
```

□

1.3 Discussion

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 [7] 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. [12] Chapter 10,
- instead of passing classes as parameters one can extend abstract inner classes which serve as counterparts of formal parameters [13] p.176,
- provides a convenient way to express call back objects [12],
- 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 [1], [12],
- allows to inherit protocols [1] p.112-113,
- enables inheritance of a class put earlier into a tree-like library of classes,
- and many others.

Chapter 2

Formulation of the problem

2.1 Preliminaries

We assume that the reader is accustomed to the programming language Java [7] 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 additional class *Object*. Now, we strip the program[1] and leave only the clauses

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

and the corresponding closing braces *}*.
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.[2]

Definition 2.1.1. Qualified type*, is a finite sequence of class identifiers separated by dots.*

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 extension type associated to it. For some classes the type is empty (in these cases the keyword **extends** is omitted – it means that that for these classes the direct superclass is the predefined class *Object*), for some other classes the type consists of one class identifier, for other classes the type is a qualified type. If a type consists of just one identifier then it is the name of the direct

[1] i.e. we throw away all instructions and all declarations other than class declarations

[2] 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 [11]. For example, A.B.F denotes the class F declared inside the class B which is declared in a top level class A.

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.($\star$). 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) [3] 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** extended by a special empty type ε see function ext below,
- $decl : Classes \cup \{Object\} \longrightarrow Classes \cup \{Root\}$
  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 $ext(K) = \varepsilon$. This empty type will denote a special class *Object* as a direct superclass of class K.

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

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

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 $\iff (\exists Y)(decl(Y) = K \wedge C = name(Y))$. The

[3]"Attribute" is the jargon of Simula67, Loglan'82 and BETA, "member" is the jargon of SMALLTALK, C++ and Java.

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. 1.1.
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 total 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.
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

$$bind : Types \times Classes \cup \{Root\} \to Classes \cup \{Object\}$$

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

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

*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*

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

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 $bind(C\ in\ K)$ is undefined in the remaining cases.*

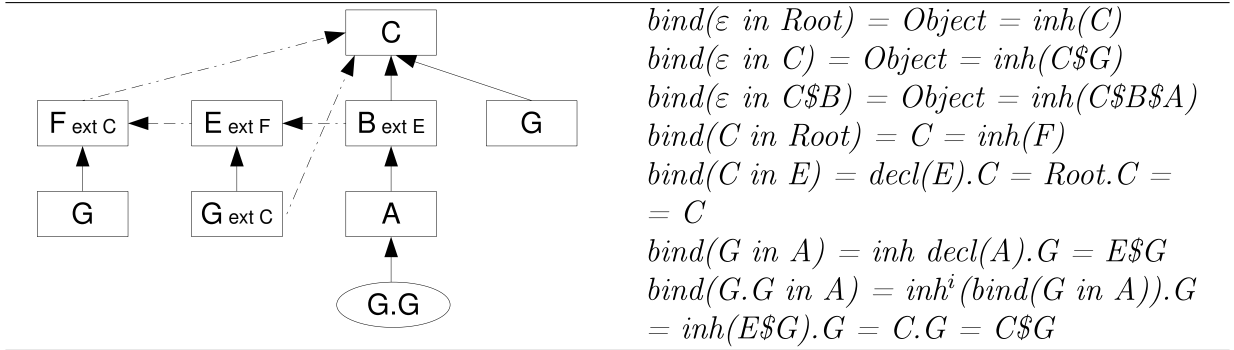
*B) (*inductive step *) Let $X \neq \varepsilon$. For any class K the meaning of a type of the form $X.C$ in the class K is determined in two steps.*

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

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

Compare our definition with the text of JLS [7](Sections 6.3, 6.5.5, 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. 2.2.

Example 2.1.3. *This example ilustrates our definition of the function bind.*



$bind(\varepsilon\ in\ Root) = Object = inh(C)$
$bind(\varepsilon\ in\ C) = Object = inh(C\$G)$
$bind(\varepsilon\ in\ C\$B) = Object = inh(C\$B\$A)$
$bind(C\ in\ Root) = C = inh(F)$
$bind(C\ in\ E) = decl(E).C = Root.C =$
$= C$
$bind(G\ in\ A) = inh\ decl(A).G = E\$G$
$bind(G.G\ in\ A) = inh^i(bind(G\ in\ A)).G$
$= inh(E\$G).G = C.G = C\$G$

□

The following relation *dep* plays an important rôle in the further considerations. In the description of the Java [7] it is called the dependency relation (induced by a superclassing function *inh*).
Let $ext(K)$ be the following type $C_1.C_2.\ ...\ .C_i.\ ...\ .C_n$. Then $ext(K)|^i$ denotes the initial segment $C_1.C_2.\ ...\ .C_i$ of length i, of the type $ext(K)$.

Definition 2.1.4. *The dependency relation dep is*

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

The above definition can be read as follows: let a class K be of the form: **class** C **extends** $C_1.C_2.\ ...\ .C_i.\ ...\ .C_n$ { ... } then the class K depends on every class designated by the type $ext(K)|^i$. In JLS [7] (Section 8.1.4) one finds the following sentence: *It 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*.

Figure 2.1 illustrates the way of computing the value of the function *inh* and the relation *dep*.

Definition 2.1.5. *A stripped Java program P is well-formed iff its class' structure* S_P *can be extended by a funtion*

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

such that it satifies the following two conditions I_1*) for every class* $K \in Classes$ *the value* $inh(K)$ *is defined and the following equality holds*

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

I_2*) The relation dep contains no cycle .*

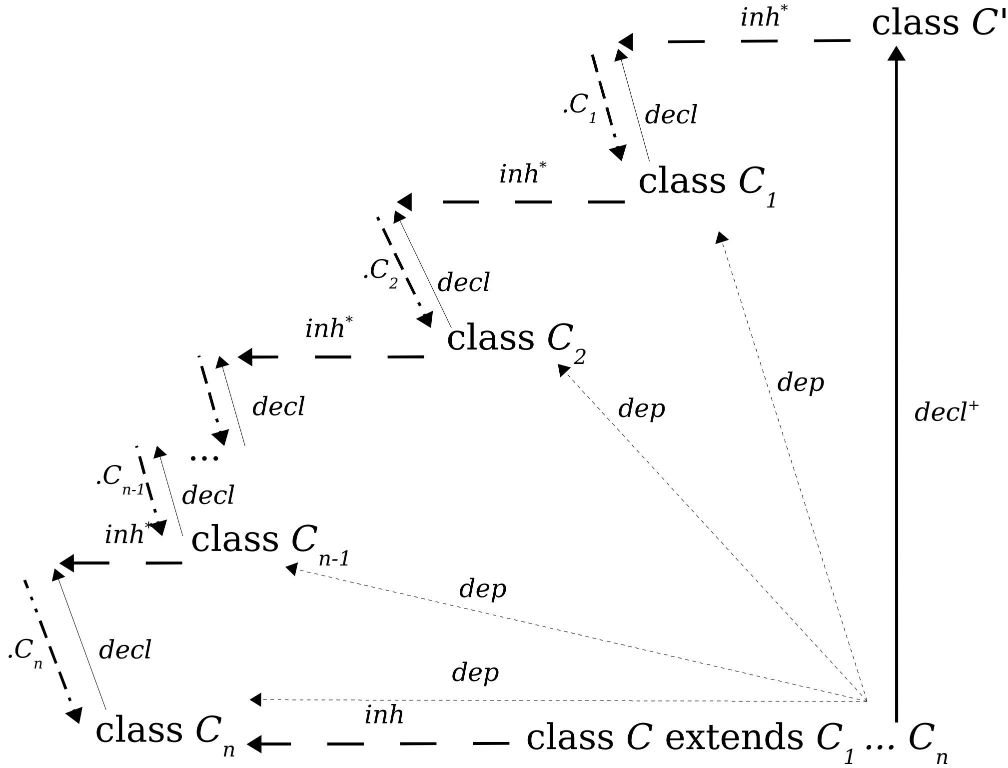


Figure 2.1: Direct superclass of **class** C **extends** $C_1.C_2.\ ...\ .C_{n-1}.C_n$.
Let K denote the class named C.
The diagram can be viewed from several angles.
First, if we delete the arrows *dep* and *decl* and keep the arrow $decl^+$, then the diagram obtained in this way commutes. This is so 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.

Now we are ready to specify the problem of determining the direct superclasses.

Problem 2.1.6. *For a given structure of classes $\mathcal{S}$, find whether the structure is well formed, and if it is so compute the function inh which satisfies conditions I_1 and I_2, in the case of the negative answer signal an error.*

2.2 The algorithm computing function bind

In the following two chapters we shall use two auxiliary functions.

```
Bind(Id id, Classes ∪ {Root} K): Classes ∪ {Object, 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, Classes ∪ {Root} K): Classes ∪ {Object, 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
```

For algorithmic purposes we assume that these functions are working on a data structure $\mathcal{S}^{prog}$ beeing a sligthly modified structure $\mathcal{S}$ from the previous chapter.

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

where $Classes, Id, Types, name, ext, Root, Object$ are as previously defined,
null is a special constant representing an undefined class.
$decl^{prog}$ extends the previous function $decl$ by adding $decl^{prog}(Root) = null$
$inh : Classes \cup \{Object, Root\} \longrightarrow Classes \cup \{Object, null\}$
is a partial acyclic function defined step by step by the algorithm in the next section.

Next we modify the function $.C$ defined in the previous chapter. We extend it from partial to the total function $.C$

$$.C : Classes \cup \{Root, Object\} \longrightarrow Classes \cup \{Object, null\}$$

by putting $K.C = null$ for all pairs for which the former value of this function was undefined.

For the readers convenience the upper subscript *prog* was omitted in the text of programs as it is clear which function should be taken into account.

Now we are ready to give a program which defines the function bind_{prog} for the purpose of the next section.

```
bind_prog(Types Path in Classes ∪ {Root} M): Classes ∪ {Object, null}
    var Class ∪ {null} L
    if length(Path) = 0 then return Object;
    L := Bind(Path[1], M);
    if L = null then return null;
    for i := 2 to length(Path) do
        L := BindInh(Path[i], L);
        if L = null then return null;
    endfor;
    return L;
end bind_prog
```

Lemma 2.2.1. *For every class K if K = bind(T in M) for some type T and class M then also* $bind_{prog}$*(T in K) = M.*

Chapter 3

Non-deterministic algorithm

3.1 The algorithm

Below, we present a non-deterministic, abstract algorithm named LSWA, which computes the function *inh*. The algorithm uses function *bind* defined in the previous section as $bind_{prog}$ which satisfies the corresponding lemma 2.2.1
Data Structure: The class structure $\mathcal{S}^{prog}$ of a program.
Argument: The graph G representing the structure $\mathcal{S}^{prog}$.
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 2.1.6.

Algorithm LSWA:

$Visited := \{Root,\ Object\};$
$inh := \{(Root, null),\ (Object, null)\};$
while $Visited \neq (Classes \cup \{Root, Object\})$
do
  $Candidates := \{K : decl(K) \in Visited \wedge K \notin Visited\}$
  if $(\exists K \in Candidates)\ bind(ext(K)$ in $decl(K)) \in Visited$
  then
    let K be a $Candidate$ taken arbitrary from the above test ;
      {this choice is made in the nondeterministic way}
    $M := bind(ext(K)\ in\ decl(K));$
    $inh := inh \cup \{\langle K, M\rangle\};$
    $Visited := Visited \cup \{K\}$
  else
    Error
  endif
endwhile

Observe that the command "let ..." is instruction of non-deterministic choice. The values of function *bind* are computed using the current diagram of function *inh* computed so far. For the algorithm of bind consult section 2.2

The word *Error* is an abbreviation of the following instruction

if $\exists K_{K\in\ Candidates} \exists i_{1\leq i\leq length(ext(K))} \forall j_{1\leq j<i} (bind(ext(K)|^j\ in\ decl(K)) \in Visited$
$\wedge\ bind(ext(K)\ |^i\ in\ decl(K))$ 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 $//\forall K_{K\in\ Candidates} \exists i_{1\leq i\leq length(ext(K))} \forall j_{1\leq j<i} (bind(ext(K)|^j\ in\ decl(K))$
$// \in Visited\ \wedge\ bind(ext(K)\ |^i\ in\ decl(K)) \in Candidates)$
throw new Signal_ConditionI_2_violated() //there is a cycle in the *dep*-relation
endif

We shall prove:

- This non-deterministic algorithm is determinate, it means that for every data, i.e. a structure of classes, any two acceptable computations of the algorithm give the same result.
- The algorithm either
  computes the correct solution, i.e. the function *inh* satisfying conditions I_1 and I_2
  or
  it adequately signals that no solution exists for the given structure of classes.

3.2 Analysis of the algorithm

3.2.1 Correctness

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 3.2.1. *(*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 3.2.2. *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 $Classes \cup \{Root, Object\}$.

Remark 3.2.3. *For every state S, the graph*

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

is a tree with the root Root. Graph

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

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 3.2.4. *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* $1 \leq i < length(P)$.

Proof. Assume the thesis of the lemma is wrong. Then there is a greatest i_0 such that $1 \leq i_0 < length(P)$ 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 3.2.3 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 3.2.3 since $C_{i_0+1} \in Visited_S$ then also $C_{i_0} \in Visited_S$. Contradiction. □

Lemma 3.2.5. *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* $i \geq 0$: $inh_S^i(K) = inh^i(K)$ *or both sides are undefined.*
B) For every class $K \in Visited_S$ *and for every type* P :
if for every $1 \leq i < length(P)$, $bind_{inh_S}(P|^i \text{ in } K) \in Visited_S$ *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 3.2.3. 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, $n \geq 1$. Let us consider type $P.C$. From the assumptions of this lemma we have

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

Therefore $bind_{inh_S}(P|^i \text{ in } K) \in Visited_S$ for $1 \leq i < length(P)$. 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 3.2.6. *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 3.2.4 we have $bind_{inh_{S_q}}(ext(K)|^i$ in
$decl(K)) \in Visited_{S_q}$ for $i = 1, ..., length(ext(K)) - 1$. Let
$M \overset{df}{=} bind_{inh_{S_q}}(ext(K) \text{ in } decl(K))$. 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 3.2.7. *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 3.2.8. *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 3.2.7 $inh_{S_1} \subseteq inh_{S_2} \subseteq inh_S$. From the algorithm the class $bind_{inh_{S_1}}(ext(K) \text{ in } decl(K)) \in Visited_{S_1}$. Then, by Lemma 3.2.4 for every $1 \le i < length(ext(K))$ the class $bind_{inh_{S_1}}(ext(K)|^i \text{ in } decl(K)) \in Visited_{S_1}$. Now by the Lemma 3.2.5
$bind_{inh_S}(ext(K) \text{ in } decl(K)) = bind_{inh_{S_1}}(ext(K) \text{ in } decl(K))$
$bind_{inh_{S_1}}(ext(K) \text{ in } decl(K) = inh_{S_2}(K)$ (see above)
$inh_{S_2}(K) = inh_S(K)$ (by the above inclusion).
This ends the proof of property I_1. □

Lemma 3.2.9. *With the assumptions of the previous lemma we observe that if there exists a cycle in the relation $deps_S$ 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 $deps_S$.
Let $V = \{\overleftarrow{K_1 \rightarrow K_2 \rightarrow\ \dots\ \rightarrow K_p}\}$ be this cycle. I.e. for $j = 1, ..., p-1$ 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 $1 \le i \le length(ext(K_1))$. 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 3.2.4, 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 3.2.5 $bind_{inh_{S_0}}(ext(K_1)|^i \text{ in } decl(K_1)) = bind_{inh_S}(ext(K_1)|^i \text{ in } decl(K_1)) = K_2$. 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 3.2.10. *(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.*

3.2.2 Completeness

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 3.1. 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 3.2.11. *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.*

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

The algorithm terminates erroneously in final state

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

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 3.2.12. *We consider the instructive Example 1.2.3 again. It is showing up the second case. Our algorithm terminates erroneously in final state shown on Fig.3.1.*
A and B are the two candidates. The value bind(A in Root) is A, the value bind(B in Root) is B. □

Example 3.2.13. *This example has exactly one solution inh which fulfills I_1. inh has no cycles, but its dependency relation dep has a cycle.*

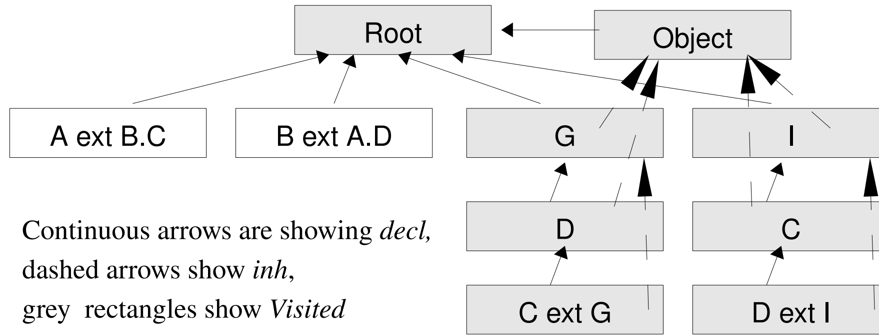


Figure 3.1: Final state for Example 3.2.12 (and Example 1.2.3)

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 $A \stackrel{dep}{\longrightarrow} B \stackrel{dep}{\longrightarrow} A$.
This example is so instructive also because it is showing that Igarashi's and Pierce's so called sanity conditions 6) and 7) in [11] are more liberal than the condition "no class depends on itself" *taken from JLS [7]. Condition 6) is saying that inh has no cycles. Condition 7) is saying that no class A is inheriting its own inner class (no* $A \stackrel{inh^+}{\longrightarrow} B \stackrel{decl^+}{\longrightarrow} A$ *is allowed)".*
□

The example 3.2.11 motivates the following

Definition 3.2.14. *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* $1 \leq i \leq length(ext(K))$, $decl(K) \in Visited_S$ *and* $bind_{inh_S}(ext(K)|^i$ *in* $decl(K))$ *is undefined and for all* $1 \leq j < i$ $bind_{inh_S}(ext(K)|^j$ *in* $decl(K)) \in Visited_S$.

Lemma 3.2.15. *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 3.2.14. By Lemma 3.2.6, $inh_S \subseteq inh$. Moreover, for all $1 \leq j < i$ $bind_{inh_S}(ext(K)|^j$ in $decl(K)) \in Visited_S$. The values $inh(Root)$ and $inh(Object)$ are undefined. Observe that $decl(K) \in Visited_S$.
By Lemma 3.2.5 we have $\forall_{M \in Classes}(bind_{inh_S}(ext(K)|^i \text{ in } decl(K)) = M \Leftrightarrow bind_{inh}(ext(K)|^i \text{ in } decl(K)) = M)$.
Since inh satisfies I_1, $bind_{inh}(ext(K)$ in $decl(K)) = inh(K)$ is defined. Hence the right-hand 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 analyse 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 3.2.16. *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 3.2.17. *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 $1 \leq i \leq length(ext(K))$, $M = bind_{inh_S}(ext(K)|^i \text{ in } decl(K))$ and for all $1 \leq j < i$, $bind_{inh_S}(ext(K)|^j \text{ in } decl(K)) \in Visited_S$.*

Proof. Let $I_K = \{l : 1 \leq l \leq length(ext(K)) \wedge (\forall 1 \leq j < l)\ bind(ext(K)|^j \text{ in } decl(K)) \in Visited_S\}$. 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 $(\forall 1 \leq j < l)\ \ bind(ext(K)|^j \ \text{ in } \ decl(K)) \in Visited_S$ 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 $bind_{inh_S}(ext(K)|^i \text{ in } decl(K))$ 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 $M = bind_{inh_S}(ext(K)|^i \text{ in } decl(K))$. 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 $M = bind_{inh_S}(C \text{ in } decl(K))$. By the definition of $bind_{inh_S}$ $M = (inh^l_S(decl^k\ decl(K))).C$ for some l and k. Hence $decl(M) = inh^l_S(decl^k\ decl(K))$. Since $K \in Candidates_S$ we know $decl(K) \in Visited_S$. From Remark 3.2.3 we obtain $decl(M) \in Visited_S$.
Consider the case $i > 1$. From the definition of i we have
$bind_{inh_S}(ext(K)|^{i-1} \text{ in } decl(K)) \in Visited_S$. Let $P = ext(K)|^{i-1}$ then $ext(K)|^i = P.C$. Since $M = bind_{inh_S}(P.C \text{ in } decl(K))$ then by definition of $bind_{inh_S}$ we obtain

$$decl(M) = inh^l_S(bind(P \text{ in } decl(K)) \qquad \text{for some } l \geq 0.$$

Now we apply Remark 3.2.3 and obtain $decl(M) \in Visited_S$. □

Corollary 3.2.18. *Suppose that the assumptions of lemma 3.2.17 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 $1 \leq j < i$ $bind_{inh_S}(ext(K)|^j \text{ in } decl(K)) \in Visited_S$. Suppose that a function inh satisfying I_1 exists.
Then $M = bind_{inh}(ext(K)|^i \text{ in } decl(K))$.

Proof. By the lemmas 3.2.5 and 3.2.6. □

Lemma 3.2.19. *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 3.2.15) then for any natural number n one can find a sequence $K_1 \overset{dep}{\to} K_2 \overset{dep}{\to} ... \overset{dep}{\to} K_n$, 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 $K_1 \overset{dep}{\to} K_2 \overset{dep}{\to} ... \overset{dep}{\to} K_n$ which satisfies the thesis of the lemma, i.e. $K_n \in Candidates_S$. By Lemma 3.2.17 and Corollary 3.2.18 there exists a class $M = bind_{inh}(ext(K_n)|^i$ 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 3.2.20. *(*on completeness*) The algorithm signals Error iff 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 3.2.15) and consequently inh does not enjoy the property I_1 or (by Lemma 3.2.19) there exists a sequence $K_1 \overset{dep}{\to} K_2 \overset{dep}{\to} ... \overset{dep}{\to} K_n$ 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 2.1.2. □

Chapter 4

A deterministic algorithm

4.1 The algorithm

This section presents a deterministic algorithm which elaborates direct superclasses. This algorithm has several advantages over the non-deterministic one of c.f. section 3.1

- 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 coloring the visited classes i.e. nodes of classes structure $\mathcal{S}$. 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, path (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.

In order to define proper data structure for the algorithm we extend the structure $\mathcal{S}^{prog}$ defined in section 2.2 by the function $sons : Classes \cup \{Root\} \longrightarrow OrderedCollectionofClasses$ Members of the ordered collection sons(K) are defined by the follwing equivalence:

$$M \in sons(K) \equiv decl(M) = K$$

The function *sons* together with its the order of its result collection is produced (like other elements of the structure $\mathcal{S}^{prog}$) as result of syntax analysis. The simplest ordering is that induced by the textual order of classes declarations. The ordering of a collection sons(K) in relevant in the execution of a loop **for** (Classes L : sons(K))
The algorithm uses functions Bind and BindInh defined in section 2.2, and consists of a few procedures.
Data Structure: The class structure $\mathcal{S}^{prog}$ of a program and the function sons
Argument: The graph G representing the structure $\mathcal{S}^{prog}$ and the function sons.
Result: The function inh which for each class $A \in Classes$ shows a class B, the direct superclass of class A, such that inh satisfies problem 2.1.6 or signaling an error if such function inh does not exist. In the case when error is signalled error recovery is performed in order to define inh to the rest of remaining classes.

```
Algorithm
    DCStack := EmptyStack
        {this stack is a global structure and will be used
          for diagnostics of a cycle in dep relation}
    Mark Root and Object White and all other classes mark Black
        {Root and Object have inh undefined and all other classes were not processed yet}
    Let inh(K)=null for every K ∈ Classes
    call Preorder(Root)
end Algorithm

Preorder(Classes K)
    if K is Black then
        {we proceed only with such K which was not processed yet by ComputeInhFor
         called form itself}
        call ComputeInhFor(K)
    endif
    for (Classes L : sons(K)) do
        call Preorder(L)
    endfor
end Preorder

CycleMessage(Classes K)
{dump DCStack in order to write out the whole dependency cycle}
    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 Classes ∪ {null} L i.e. value of variable L is either a Class or null
        {variable L is used to compute consequtive types found in ext(K)}
    { decl(K) is White }
    {decl(K) was yet processed and has its inh computed}
    Mark K Grey
    {denotes the presence of K in the stack of ComputeInhFor activation records and
     is used for detection of cycle in dep relation}
    if length(ext(K)) =0 then
        L := Object;
        {empty type denotes, by default, inheritance form Object}
    else
        i:=1
        L:=Bind(ext(K[i], decl(K))
        {L is the meaning of the first identifier in ext(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", header(K))
            inh(K):=Object; Mark K White – – recovery
            return
        endif
        if L is Grey then
            {L in stack of ComputeInhFor – cycle in dep detected}
            Push(DCStack, L); Push(DCStack, ext(K)|^i)
            Mark K Black; Mark L Red
                mark red the bottom element of the cycle in stack of ComputeInhFor
            if K=L then – – one element cycle detected
                call CycleMessage(K) – – diagnostics
                L := Object – – recovery
                exit
            endif
            return
                {otherwise the cycle is longer and we should continue popping
                 ComputeInhFor stack in order to find the red bottom of the cycle}
        endif
        if L is Black then
            {L was not processed yet}
            call ComputeInhFor(L)
                {compute inh for L by demand}
            if not empty(DCStack) then
                {L was involved in dep cycle, so push it to DCStack}
                Push(DCStack, L) ; Push(DCStack, ext(K)|^i)
                if K is Red then
                    {bottom of the cycle find}
                    call CycleMessage(K) – – produce diagnostics
                    L:=Object – – recovery
                    exit
                endif
                Mark K Black
                {bottom of the cycle lies below current ComputeInhFor activation record
                  K will be treated as unprocessed for further analysis
                  and will be pushed onto DCStack immediately after return}
                return
            endif – – ComputeInhFor has returned with the proper value of inh(L)
        endif – – the value of inh is properly set for l
        {Assertion1:  L=bind(ext(K)|^i in decl(K)) and L is White }
        if i=length(ext(K)) then exit endif
            {if the value of L is the whole type ext(K) exit while
              otherwise compute the next element of ext(k)}
        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 }
    {L is either the proper value of ext(K) for nonempty type
      or is set to Object by error recovery}
endif
{Assertion3: L=bind(ext(K) in decl(K)) and L is White or K is Red and L=Object }
```

```
    {L is either the proper value of ext(K) or is set to Object by error recovery}
        inh(K):=L ; Mark K White
end ComputeInhFor
```

4.2 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.

4.2.1 Experiments

We did some experimenting before proceding to wording of lemmas and proving them. Two animations of experiments are offered at the URL:
http://duch.mimuw.edu.pl/∼ salwicki/CSP2007/
presentationLagow/prezentacja21Listop2007.pdf.
Here we quote a slide of detection of cycle in dep relation.

4.2.2 Proof of correctness

Lemma 4.2.1. *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.

(iv) In procedure $ComputeInhFor$ node K may get color Black. It is done 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.

(vi) The procedure ComputeInhFor will never be called with the parameter K equal Root or Object.

Proof. (i) Look at the code of $ComputeInhFor$ and check that there are only two places in the algorithm where a node K is marked White. In both cases this instruction is accompanied by the instruction of the form $inh(K) := expression$.

(ii) Check the code of $ComputeInhFor$.

(iii) It follows from the previous observations.

(iv) Check the code of $ComputeInhFor$.

(v) Obvious.

(vi) This is obvious consequence of the fact that Root and Object are White and all procedure calls of ComputeInhFor have the argument K Black. As the consequence we observe that the expressions of the form ext(K) have always the well defined value.

□

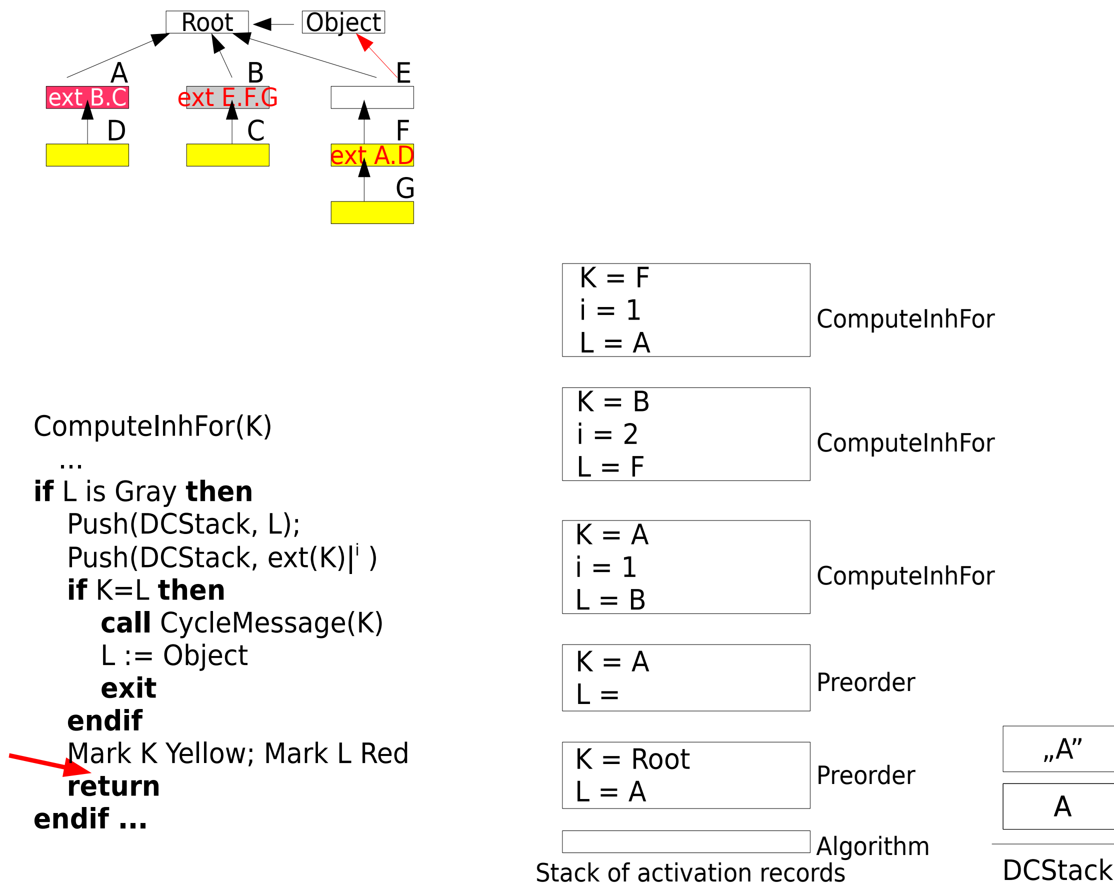


Figure 4.1: The cycle of dep relation of dependency is: A *dep* B *dep* F *dep* A
Explanation of the diagram:
- in left upper corner you see the structure of classes,
- in right lower corner the content of DCStack,
- to the left the stack of activation records is shown,
- in left lower corner we exhibit a part of code of currently executed method.
Relation dep is visible in activation records of ComputeInhFor
the node K *depends* on node L.
Attention! This picture uses colour *Yellow* instead of colour *Black* – it was done in order to avoid blackening the rectangles and texts in them, simply read Yellow as Black.

The next lemma analyzes actions taken when a node is marked Red. The condition necessary to mark a node Red is as follow: 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 activation record of ComputeInhFor where the parameter K is equal to the recently computed value of the variable L. The algorithm pushes two elements on DCStack. Having this in mind we formulate the following

Lemma 4.2.2. *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 Grey then the algorithm puts two elements onto DCStack and marks K Black, then the activation record 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 4.2.3. *Whenever the algorithm reaches* Assertion1 *the node L is White*

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

Lemma 4.2.4. *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. The class 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 4.2.1 that the new graph is a tree again. □

Corollary 4.2.5. *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 4.2.6. *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 $decl(L)$ 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 4.2.5 we have $inh^*(K)$ is White hence $(decl \mid inh)^*(K)$ is White which ends the proof of A). □

Lemma 4.2.7. *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 4.2.6. Instruction call BindInh occurs once, inside the while statement. Its second argument L is White c.f. lemma 4.2.3. □

Lemma 4.2.8. *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 4.2.9. *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*

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

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

Lemma 4.2.10. *Assume that a computation of the algorithm terminates without signalling an error. The following formula is the invariant of the loop while in the $ComputeInhFor(K)$*

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

Lemma 4.2.11. *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 Grey. 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 4.2.12. *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 2.1.6 are satisfied.*

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

Theorem 4.2.13. *If the algorithm terminates and signals an error then no solution exists, i.e. the structure of classes is erroneous.*

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 4.2.2. If during an execution of the algorithm a message was printed "undeclared class X in the header of the class"+ *header*(Y) then no solution exists. Here X stands for a type name, and in the structure of classes, no class named X is accessible from the place of declaration of the currently analyzed class Y. For the proof see [17]. □

4.2.3 Complexity of the algorithm

As concerns the complexity of the algorithm it can be estimated as follow for the well-formed programs: Each node is visited twice: once by the procedure ComputeInhFor and second by the procedure Preorder. During these 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 include $O(n^2)$ – the cost of operations Bind executed. 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.

4.3 Remarks on efficiency and error recovery

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 [11] is highly ineffective, for it requires elaboration of each segment of a qualified type anew. The nondeterministic algorithm (see section 3.1) stores the elaborated types and makes possible the further usage of earlier stored results. The deterministic algorithm uses pebbling by pebbles of different colours thus making the algorithm more efficient.
The algorithm continues its work after discovering an error in the structure of classes. The first error found is accompanied by the unquestionable diagnostic. Hence one can confide in it. The diagnostic of subsequent errors has a lower level of credibility. Can we trust the subsequent error signals? Is it possible to prepare a better scheme of error recovery?
The compilers of Java do error recovery only in selected phases (most of them interrupts the parsing after a syntactic error found). On the other hand, the errors found during static semantic analysis are always recovered. The errors discovered by our algorithm belong to this class. We felt obliged to include such a mechanism in our deterministic algorithm. During the work on deterministic algorithm we concentrated on its correctness and completeness (i.e. diagnostic of errors) and the error recovery came as an easy extension. In the case when the structure of classes of a program is not well-formed the estimation of the cost is an open problem.

4.4 Signalling the errors

In this appendix we present a source of an erroneous Java programs and the diagnostic produced by our algorithm.

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

```
class B extends E.F.G {
    class C { }
}

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

Below is a message of our algorithm

```
Dependence cycle :
line1 : class A extends B.C {
depends on class named B which is :
line5 : class B extends E.F.G {
which in turn depends on class named E.F which is :
line10 : class F extends A.D {
which in turn depends on class named A which is :
line1 : class A extends B.C {}
```

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

Chapter 5

Remarks on earlier work

5.1 Introduction

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

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

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

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.

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 [5,6,7] the problem of maintaining the Display Vector was addressed and solved.

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.

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 3.2.13 in Section 4.1 is a drastic counter example of equivalence. Example 3.2.13 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.

The subsection 5.2.1 of the section 5 of [11] is devoted to the elaboration of types, which makes the identification of direct superclasses possible. The Table 1 cites six inference rules of the paper [11][p.35]. They define a calculus. We shall name it the IPET calculus. We are analyzing the calculus. The aim of the IPET calculus is to help in identifying the direct superclasses in any Java program. We present some observations.

1. The calculus is not determinate. It means that it is possible to derive two or more different classes as a direct superclass of certain class. One may say the calculus is inconsistent.

2. Moreover, there exist two different models of the calculus.

3. Moreover, the models do not enjoy the property: the intersection of two models is a model. Therefore it is difficult to say what is the meaning of the calculus.

The authors are aware of the non-determinacy. They say the calculus plus a metatheoretic hint: *apply the rules from bottom up* may be called an algorithm. They have chosen an inadequate word. It is not an algorithm however. For it does not enjoy the termination property c.f. [11][p. 34] . Therefore we propose to call it a *method*. Again the method may lead towards different answers. We shall show that the method can be specified in two different manners. The IPET calculus may be used to define the inheritance function *inh* from classes to classes. We can take another approach and ask: has the IPET calculus one or more models? It turns out that it has several, non-isomorphic, models. Let us remark that each model can be constructed by the corresponding algorithm. Hence it is necessary to add some hint of metatheoretical nature. Usually, a calculus (or a theory) is accompanied by the metatheoretical hint: *choose the least model*. We are going to show that this will not work. For the intersection of two models needs not be a model.

It seems that the source of the problems is in admitting an inference rule (ET-SimpEncl). One of the premises of this rule is a metatheorem: $P \vdash X.D \Uparrow$. This formula expresses the following property: for every class T there is no proof of the formula $P \vdash X.D \Rightarrow T$. One remedy would be to eliminate the rule and to replace it with some rules that do not introduce metatheoretic premises and such that the premises are positive formulas. Another approach would consists in extending the language of the theory, such that the expression $P \vdash X.D \Uparrow$ were a well formed expression of the language, and adding some inference rules to deduce formulas of this kind. Nothing of this kind happens in [11].

We are stressing the fact that the discussed paper is one of many papers of various authors, where the reader discovers a metatheorem as a premise of an inference rule. Therefore our remarks are of general character.

5.2 Igarashi's and Pierce's calculus IPET for elaboration of types

Igarashi and Pierce [11][*5.2.1* pp.34-36] are presenting a calculus IPET of derivation rules for a so called elaboration relation of types. The formulae of the calculus have the form (are written as) $\boxed{\texttt{P} \vdash \texttt{X} \Rightarrow \texttt{T}}$ to be read: *The simple or qualified class type* `X` (i.e. a non-empty sequence of class identifiers separated by periods) *occurring inside the directly enclosing body of class declaration occurrence* `P` *is elaborated to* (resp. is bound to) *the class declaration occurrence* `T`. In other words: the meaning of the type `X` in the class `P` is the

class T. We have to differentiate between a syntactical entity and its occurrences, see the ALGOL68-report [14], because a class declaration (class for short) may occur several times in a given program .

Observe that there is a bijection between class occurrences like P (or T) and their so called absolute types (paths) $\mathtt{C}_1.\cdots.\mathtt{C}_n$ where $\mathtt{C}_n$ is the name of class P and $\mathtt{C}_{n-1},\cdots,\mathtt{C}_1$ are the names of the successive class occurrences which enclose class occurrence P. To understand this one should remark that the classes of a program form a tree. The root of the tree is a fictitious class that encloses all the top level classes of the program. Let n be an internal node of the tree. It can be identified with the path leading from the root to it. Such a path consists of the names of classes. All direct inner classes declared in the class which is the node n are the sons of the node n. Therefore we are entitled to identify an occurrence of a class declaration and the absolute path of it. Beside the user declared class occurrences in a Java-program there are two implicit, fictitious class occurrences:

1. *Root* $= \{\cdots\}$ which is enclosing all top level classes (i.e. other class occurrences) of the Java-program and which has no name nor extends clause. The authors of [11] represent *Root* by its fictitious name $\star$ which users are not allowed to write. $\star.\mathtt{C}_1.\cdots.\mathtt{C}_n$ is identified with $\mathtt{C}_1.\cdots.\mathtt{C}_n$.

2. *Object* $=$ `class Object` $\{\cdots\}$ the name of which is `Object`, which is directly enclosed by *Root* and which has also no extends clause. Without loss of generality we can assume that there are no classes declared inside the body of *Object*.

Let us explain the meaning of some premises in the inference rules. In three rules one finds the premise of the form $CT(P.C) =$ **class** C **extends** X $\{\cdots\}$. In this way the authors Igarashi and Pierce express the fact that the class $P.C$ extends the type X i.e. the class which is the meaning of the type X in this place where the declaration the class $P.C$ occurs. The formulas of the form $P.C \in Dom(CT)$ mean the program contains the class named C in its directly enclosing class to be identified with the path P. Obviously, the formula of the form $P.C.D \notin dom(CT)$ expresses the fact that the class to be identified with the path $P.C$ does not contain any class named D. In the Table 1 we present Igarashi's and Pierce's calculus IPET for elaboration of types.
Below we collect some observations and comments.

1. The system IPET is inconsistent! Consider a Java program with a user declared class Object. From the axiom (I ET-OBJECT) one obtains that the meaning of the name `Object` is *Object*, or $\star$.*Object*. From the rule (II ET-IN-CT) one obtains *P.Object* where P is the class containing the user declared class `Object`.
Remedy: Consider only programs without user declared class named `Object`.

2. The rule (III ET-SimpEnc) has four premises. The fourth premise of the form $\boxed{P \vdash X \Uparrow}$ is in fact a metatheorem "*there is no class T such that the triplet $P \vdash X \Rightarrow T$ has a formal proof*". This rule is a source of severe problems as we shall see below.

3. There is **no** definition of the notion of proof in the system IPET of inference rules. Should one accept the classical definition of the notion of formal proof then the lack of possibilities to derive premises of the form $P \vdash X \Uparrow$ becomes evident. We know, the standard answer to this remark is: *but everything is finite and therefore one can control the situation.* Is this one person added to the definition of the proof? What instructions are given to her/him making the task of recognition of the impossibility of the proof possible?

Table 5.1: Igarashi's & Pierce's rules of elaboration

Rule	
I. (ET-Object)	$P \vdash \texttt{Object} \Rightarrow Object$
II. (ET - In CT)	$\dfrac{P.C \in dom(CT)}{P \vdash \texttt{C} \Rightarrow P.C}$
III. (ET-SimpEncl)	$\dfrac{P.C.D \notin dom(CT) \qquad P \vdash D \Rightarrow T \qquad CT(P.C) = \textbf{class}\ \texttt{C}\ \textbf{extends}\ \texttt{X}\ \{\cdots\} \qquad P \vdash X.D \Uparrow}{P.C \vdash D \Rightarrow T}$
IV. (ET-SimpSup)	$\dfrac{P.C.D \notin dom(CT) \qquad CT(P.C) = \textbf{class}\ \texttt{C}\ \textbf{extends}\ \texttt{X}\ \{\cdots\} \qquad P \vdash X.D \Rightarrow T}{P.C \vdash D \Rightarrow T}$
V. (ET-Long)	$\dfrac{P \vdash X \Rightarrow T \qquad T.C \in dom(CT)}{\texttt{P} \vdash \texttt{X.C} \Rightarrow \texttt{T.C}}$
VI. (ET-LongSup)	$\dfrac{P \vdash X \Rightarrow P'.D \qquad P'.D.C \notin dom(CT) \qquad CT(P'.D) = \textbf{class}\ \texttt{D}\ \textbf{extends}\ \texttt{Y}\{\cdots\} \qquad P' \vdash Y.C \Rightarrow U}{P \vdash X.C \Rightarrow U}$

4. A reader may hesitate what does the following sentence mean "*A straightforward elaboration algorithm obtained by reading the rules in a bottom-up manner might diverge.*" [11][p.34] Two questions appear immediately. Is there an implicitly defined elaboration algorithm? What does it mean "reading the rules in a bottom-up manner"?
   Our *first guess* is that the authors think of Gentzen-style proofs. The textbook on mathematical logic [15] describes an algorithm constructing a formal proof of a logical formula. The system of inference rules must enjoy some properties and the algorithm must precisely describe which rule is to be applied in every step of the algorithm.
   Our *second guess* is as follow: Subcase 1. Consider an open question $\boxed{P \vdash X \Rightarrow ?}$ and apply the rules trying to construct the formal proof of some triplet $P \vdash X \Rightarrow T$. Depending on the rule applied we create some new open questions. In this way a tree is constructed with the nodes decorated by open questions or axioms. Once an inner node has all its sons decorated by closed triplets, one can close an open question by application of the rule that constructed the sons of the current node. Should we come back to the root with an answer the formal proof is constructed and the searched class T is found. Subcase 2. As previously consider an open triplet and build a formal proof of some triplet $P \vdash X \Rightarrow T$ applying the rules from the sixth to the first one.
   Which guess is a proper one?

5. The authors are aware that construction of the proof is not always possible. They give an evidence of the fact that the algorithm we guessed may loop without exit [11][p.34].

6. In fact the task of type elaboration is divided in two subtasks: a) to find if the program is a well formed one, b) to define a function *inh* which for every user declared class C returns the direct superclass of C. It is evident that the IPET does not help in detecting the possible errors in typing.

7. Seeing the incompleteness of the IPET calculus (c.f. the rule III) one may ask a slightly different questions: is it true that the IPET has exactly one model? We shall see that there are several models.

8. The next question appears: Is it possible to equip the calculus with an extra hint of the kind: consider the least one of all models as THE model of the IPET calculus.

9. This hope should be abandoned in the light of the section 5.5.

5.3 Langmaack's, Salwicki's and Warpechowski's binding functions $bind_{inh}$ compared with IPET

In [16] Langmaack, Salwicki and Warpechowski developed binding functions $bind_{inh}$ which, based on given inheritance(i.e. direct superclassing) functions inh, determine the associated class occurrences T for class types X directly enclosed by the bodies of class occurrences $P: \quad bind_{inh}(X \; in \; P) = T$.
Above that the authors developed an algorithm LSWA which determines superclassing function inh_0 which is the least fixed point of the continuous functional $Bdfl'(inh)$
$inh_0 = Bdfl'(inh_0)$
and is totally defined for all (finitely many) user declared classes P in a given well-formed Java-program. Especially: inh_0 satisfies the so called inheritance condition I_1, i.e. for all user declared classes $P \quad inh_0(P)$ is defined and the equation

$$inh_0(P) = bind_{inh_0}(X \; in \; P')$$

is holding where X is the type $ext(P)$ in the extends clause of P and P' is that class occurrence $decl(P)$ which directly encloses P. Because both theories of [11] and of [16] ought to agree
the ternary relation $bind_{inh_0}(X \; in \; P) = T$
should satisfy all six rules of the types elaboration relation
$\mathtt{P} \vdash \mathtt{X} \Rightarrow \mathtt{T}$
in calculus IPET. Both theories would agree perfectly iff there were exactly one distinguished satisfying binding function for a well-formed Java-program such that the set of all derivable triplets (`X`, `P`, `T`) is exactly the binding function $bind_{inh_0}$. In order to have an easier way of comparison we translate the rules to the mode of expression in [16] what is yielding the formulation of the definition 5.3.1

Definition 5.3.1. *The calculus BIPET is defined by the rules of the Table 2.*

Theorem 5.3.2. *The function $bind_{inh_0}$ is satisfying all six rules of BIPET (and hence of IPET) calculus.*

Proof. Are these six rules (implications) really holding? We shall check them and find that the answer is: Yes.
I. (BET-Object)
As *Object* is the only class named `Object` and is directly enclosed by *Root* the required equation is holding independently of all possible inheritance or direct superclassing functions inh which parametrize $bind_{inh}$.
II. (BET-InCT)
If the class P contains a direct inner class named C, i.e. $P.C$ is defined,$P.C \in Dom(CT)$,

Table 5.2: IPET rules translated using *bind* function

Rule	
I. (BET-Object)	$bind_{inh_0}(\texttt{Object} \text{ in } P) = Object.$
II. (BET - InCT)	$\dfrac{\text{the class P has the direct inner class named C,}}{bind_{inh_0}(C \text{ in } P) = P.C.}$
III. (BET-SimpEncl)	$\dfrac{bind_{inh_0}(D \text{ in } P) = T, \quad \text{no class named D in P.C}, \quad bind_{inh_0}(ext(P.C).D \text{ in } P) \text{ is undefined,}}{bind_{inh_0}(D \text{ in } P.C) = T.}$
IV. (BET-SimpSup)	$\dfrac{bind_{inh_0}(ext(P.C).D \text{ in } P) = T, \quad \text{class P.C has no direct inner class named D,}}{bind_{inh_0}(D \text{ in } P.C) = T.}$
V. (BET-Long)	$\dfrac{bind_{inh_0}(X \text{ in } P) = T, \quad \text{class T has a direct inner class named C,}}{bind_{inh_0}(X.C \text{ in } P) = T.C.}$
VI. (BET-LongSup)	$\dfrac{bind_{inh_0}(X \text{ in } P) = P'.D, \quad \text{class P'.D has no direct inner class named C,} \quad bind_{inh_0}(ext(P'.D).C \text{ in } P') = U,}{bind_{inh_0}(X.C \text{ in } P) = U.}$

then the conclusion *the meaning of the name C in the class P is the class $P.C$* is holding independently of all possible inheritance functions.
III. (BET-SimpEncl)
Let P be a user declared class. From the first premise $bind_{inh_0}(D \text{ in } P) = T$ we have that there exist natural numbers $i \geq 0,\ l \geq 0$ such that $T = inh_0^i(decl^j(P)).D$ where the pair $\langle l, i\rangle$ is the least in the lexicographic order such that the right hand side expression is defined.
The third premise "$bind_{inh_0}(ext(P.C).D \ in \ P)$ is undefined" says that for every $l \geq 0$ the expression $inh_0^l(bind_{inh_0}(ext(P.C) \text{ in } P).D$ is undefined, and so $inh_0^l(inh_0(P.C)).D$ is undefined, because the function inh_0 enjoys the property $I_1 :\ inh_0(P.C) = bind_{inh_0}(ext(P.C) \text{ in } P)$. We claim that the pair $\langle j+1, i\rangle$ is the least pair in the lexicographic order such that $inh_0^i(decl^{j+1}(P.C)).D$ is defined. Suppose that there exists a pair $\langle k, l\rangle$ such that the the expression $inh_0^l(decl^k(P.C)).D$ has a defined value and that the pair $\langle k, l\rangle$ precedes the pair $\langle j+1, i\rangle$. From the second and third premise we know that $k \neq 0$. In other words the path $inh_0^l(decl^k(P.C)).D$ goes from the class $P.C$ through the class P further on. From the previous considerations we know the pair $\langle j, i\rangle$ is the least in the lexicographic order such that the expression $inh_0^i(decl^j(P)).D$ is defined. Hence $k = j+1$ and $l = i$. The cases where $P = Object$ or $P = Root$ can be checked in a separated way.
IV. (BET-SimpSup)
Let $P.C$ be a user declared class. The premise $bind_{inh_0}(ext(P.C).D \text{ in } P) = T$ tells us that there exists a natural number $i \geq 0$ such that the value of $T = inh_0^i(bind_{inh_0}(ext(P.C) \text{ in } P)).D$ is defined. Since the function inh_0 enjoys the property I_1 we know that the expression $inh_0^i(inh_0(P.C)).D$ has a value T. Hence $T = inh^{i+1}(decl^0(P.C)).D = bind_{inh_0}(D \text{ in } P.C)$. For the pair $\langle 0, i+1\rangle$ is the least pair such that the value of $inh^{i+1}(decl^0(P.C)).D$ is defined. The only less candidate $\langle 0, 0\rangle$ is excluded by the second premise. If $P = Object$ or

$P = Root$ then the conclusion is holding trivially.
V. (BET-Long)
The conclusion is holding due to definition of $bind_{inh_0}$.
VI. (BET-LongSup)
In case $P'.D$ is user declared we begin with the third premise $bind_{inh_0}(X.C \text{ in } P') = U$. This means that there exist a natural number $l \geq 0$ such that $inh_0^l(bind_{inh_0}(X \text{ in } P')).C = U$ and l is the least one that the left hand side is defined. Making use of the first premise and condition I_1 for $P'.D$ we obtain $inh_0^{l+1}(P'.D).C = U$. From the second premise we conclude that the exponent $l+1$ is the least one such that the left hand side is defined. The soundness of this rule follows from the definition 2.1.2 of the function $bind_{inh_0}$ and from the first premise.
In case $P'.D$ is not user declared the conclusion is holding trivially.

□

Hence the binding function $bind_{inh_0}(X \ in \ P) = T$ satisfies all six rules of IPET where the given Java-program is well-formed. We recall that its least fixed point is the inheritance function inh_0 and satisfies the conditions I_1 and I_2 of the Java Language Specification JLS [7] as formalized in [16]. Our next question is: Is IPET defining $bind_{inh_0}$ uniquely? Are there other binding functions which satisfy all rules of IPET? Is there a distinguished binding function of IPET? In what sense does IPET define a distinguished binding function? Or, perhaps, there is no a reasonable way to distinguish a good inheritance function? We have already seen that IPET does not allow straightforward, constructive top-down application.

5.4 On another binding function $Bind_{inh_{B0}}$ which satisfies all rules of IPET

It is astounding that there is a way to define a family of binding functions $Bind_{inh}(X \ in \ P)$ which are different from $bind_{inh}(X \ in \ P)$ and which lead to an analogous satisfaction theorem (of IPET, by an analogous $Bind_{inh_{B0}}$ instead of $bind_{inh_0}$). As previously, we assume that every user declared class has an explicit extends clause with a type of length ≥ 1 as the authors of [11] and their calculus are requiring. $Bind_{inh}$ is to become a partial mapping

$$Bind_{inh} : Types \ \times \ \mathcal{C}^{RO} \ \xrightarrow{part} \ \mathcal{C}^{O}$$

where $\mathcal{C} = Classes$ is the set of user declared class occurrences in a given Java-program with inner classes, $\mathcal{C}^O$ is $\mathcal{C} \cup \{Object\}$ and $\mathcal{C}^{RO}$ *is* $\mathcal{C}^O \cup \{Root\}$. $Bind_{inh}$, $Types$ is the set of simple or qualified types in the Java-program. The function $Bind$is parametrized by a given partially defined inheritance function (direct superclassing function) $inh : \mathcal{C} \xrightarrow{part} \mathcal{C}^O$ as $bind_{inh}$ is. The values of $inh(Root)$ and $inh(Object)$ are undefined.

Consider the ordered alphabet $\mathcal{A}$ of the two operators inh and $decl$, where we define inh to be less than $decl$, $inh \prec decl$. This order is inducing a lexicographical (from the right), total order in the set $\mathcal{A}^*$ of all words over $\mathcal{A}$. For example, the words $inh \prec decl^\frown inh \prec decl \prec inh^\frown decl$ are in this order.
Let $w = id_1^\frown id_2^\frown \cdots id_n, \ w \in \mathcal{A}^*$. Let P be a class. The word w applied to the class P is the class $w(P) = id_1(id_2(\cdots(id_n(P))\cdots))$ or the result is undefined. Clearly $\varepsilon(P) = P$. Now, we have the following

Definition 5.4.1. *Let X be a type of length ≥ 1, $P \in \mathcal{C}^{\mathcal{R}O}$ and $inh \in \mathcal{C} \overset{part}{\longrightarrow} \mathcal{C}^O$. Then*

$$Bind_{inh}(X \text{ in } P) \overset{df}{=} \begin{cases} \mu w(P).X & \text{if } length(X) = 1 \text{ and there exists the least word } \mu w \in \mathcal{A}^* \\ & \text{such that } \mu w(P).X \in \mathcal{C}^O \text{ is defined and there are} \\ & \text{no repeated classes on this path } \mu w \text{ from } P \text{ to } \mu w(P). \\ Bind_{inh}(C \text{ in } Bind_{inh}(X' \text{ in } P)) & \text{else if } X = X'.C \text{ where } length(C) = 1 \\ & \text{and } length(X') \geq 1 \text{ and } Bind_{inh}(C \text{ in } Bind_{inh}(X' \text{ in } P)) \in \mathcal{C}^O \\ & \text{is defined} \\ undefined & \text{otherwise} \end{cases}$$

Remark 5.4.2. *It is worthwhile to observe that the $Bind_{inh}$ function defined in this way* differs *from the $bind_{inh}$ function defined earlier. Namely, in the definition of bind we consider only words of the form $inh^i decl^j$, where $i, j \geq 0$.* □

Example 5.4.3. *C.f. the class' structure of the example 5.4.4. We have*
$bind(C \text{ in } decl(B\$D)) = bind(C \text{ in } B) = inh(B).C = Bind(C \text{ in } B) = A\$C$
$bind(E \text{ in } decl(B\$D\$F)) = bind(E \text{ in } B\$D) = decl(B\$D).E = B\$E$
$Bind(E \text{ in } decl(B\$D\$F)) = Bind(E \text{ in } B\$D) = decl^\frown inh(B\$D).E = E\$A$
since $decl^\frown inh \prec decl$ □

Now we are going to look for a specific inheritance function inh_{B0} such that $Bind_{inh_{B0}}$ satisfies all rules of IPET. We go an analogous way as in [17] and look for an appropriate functional $Biddle'$ such that inh_{B0} is the least fixed point. The natural functional

$$BDfl(inh)(P) \overset{df}{=} Bind_{inh}(ext(P) \text{ in } decl(P))$$

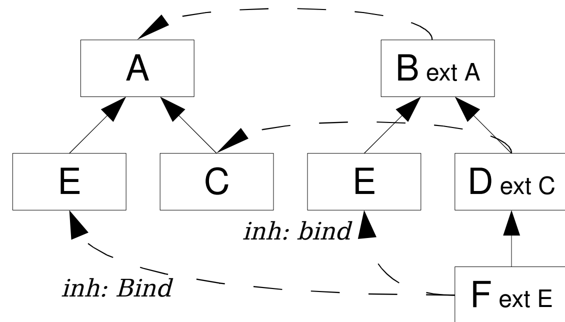
is, unfortunately, not monotone and continuous in

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

where $\mathcal{C} \overset{part}{\longrightarrow} \mathcal{C}^O$ is a cpo completely partially ordered by the set theoretic inclusion $\subseteq$ of partially defined inheritance functions with bottom function $inh_\perp = \emptyset$.

Example 5.4.4. *Let us consider the following (structure of a) Java-program:*

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



We have

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

because

$BDfl(inh_\perp)(B) = A$, $\quad BDfl(inh_\perp)(D) = \perp$, $\quad BDfl(inh_\perp)(F) = B\$E$,
$BDfl^2(inh_\perp)(B) = A$, $\quad BDfl^2(inh_\perp)(D) = A\$C$, $\quad BDfl^2(inh_\perp)(F) = A\$E$. □

Example 5.4.4 convinces us to modify functional $BDfl$ towards $BDfl'$. But first we introduce the notion of State.

Definition 5.4.5. *An inheritance function* $inh \in (\mathcal{C} \stackrel{part}{\longrightarrow} \mathcal{C}^O)$ *is called a* State *iff for all classes* $K \in dom_{inh}$ *the following two relations*

$inh(K) \in dom^O_{inh}$, *where* $dom^O_{inh} \stackrel{df}{=} dom_{inh} \cup \{Object\}\}$

$decl(K) \in dom^R_{inh}$ *where* $dom^R_{inh} \stackrel{df}{=} \{dom_{inh} \cup \{Root\}\}$

and the equation

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

are holding. □

Definition 5.4.5 is saying that $dom^{RO}_{inh} = dom_{inh} \cup \{Root,\ Object\}$ is an initial tree of the whole *decl*-tree $\mathcal{C}^{RO}$, and that for all K 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 condition I_{B1} (see Definition 5.4.7) is satisfied, restricted to dom_{inh} as a subset of $\mathcal{C}$. *inh* or its dependency relation Dep_{inh} may have cycles:

Definition 5.4.6. *The* dependency relation Dep_{inh} *associated to inh is*

$$Dep_{inh} \stackrel{def}{=} \{\langle K, Bind_{inh}(ext(K) \mid^i \ in \ decl(K))\rangle : K \in dom_{inh}, 1 \leq i \leq length(ext(K))\}$$

where $ext(K) \mid^i$ *is the initial segment of length* i *of type* $ext(K)$. □

To remind the reader(c.f. definition 2.1.5):

Definition 5.4.7. *A Java-program is called* Well-Formed *iff there exists an inheritance function* inh_{WF} *which satisfies the following two conditions*

I_{B1}) inh_{WF} *is defined for all classes* $K \in \mathcal{C}$ *and the equation*

$inh_{WF}(K) = Bind_{inh_{WF}}(ext(K) \ in \ decl(K))$

is holding for them;

I_{B2}) *the induced dependency relation* $Dep_{inh_{WF}}$ *has no cycles in* $\mathcal{C}^{RO}$. □

We consider the sub-cpo

$\mathcal{C} \stackrel{State}{\longrightarrow} \mathcal{C}^O$ of $\mathcal{C} \stackrel{part}{\longrightarrow} \mathcal{C}^O$

of inheritance functions which are States. The word "State" is chosen because our later algorithms LSWA'$_B$ resp. LSWA$_B$ which determine the least fixed point inh_{B0} are running through computation states which can be represented by the above mentioned special inheritance functions which are States.

Let's come to the desired functional $BDfl'$. Let us introduce an abbreviation $\alpha^B_{inh}(A)$ denoting the following logical formula:

$$\alpha^B_{inh}(A) : decl(A) \in dom^R_{inh} \wedge \ A \neq Root \wedge A \neq Object \wedge Bind_{inh}(ext(A) \text{ in } decl(A)) \in dom^O_{inh}.$$

The desired functional is

$$BDfl'(inh)(A) \stackrel{df}{=} \begin{cases} Bind_{inh}(ext(A) \text{ in } decl(A)) & \text{if } \alpha^B_{inh}(A) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Theorem 5.4.8. $BDfl'$ *is a monotonous functional (and consequently is continuous because* $\mathcal{C} \stackrel{State}{\longrightarrow} \mathcal{C}^O$ *is finite).*

We need four Lemmas 5.4.9 to 5.4.13 for a proof of Theorem 5.4.8.

Lemma 5.4.9. *For every State inh, for every class $K \in dom^{RO}_{inh}$ and for every type X: If $Bind_{inh}(X \text{ in } K) \in dom^{RO}_{inh}$ then $Bind_{inh}(X|^i \text{ in } K) \in dom^{RO}_{inh}$ for $1 \le i < length(X)$.*

Proof. Assume the thesis of the Lemma is wrong. Then there is a smallest i_0 with $1 \le i_0 < length(X)$ and $Bind_{inh}(X|^{i_0}$ in $K) \notin dom^{RO}_{inh}$. Then this class C_{i_0} is such that $decl(C_{i_0}) \in dom^{R}_{inh}$ because inh is a State. Then $inh(C_{i_0})$ is undefined and $Bind_{inh}(X$ in $K)$ is necessarily a nested class C_l inside or equal C_{i_0} with $decl^{l-i_0}(C_l) = C_{i_0},\ l = length(X)$, and C_l is necessarily $\notin dom^{RO}_{inh}$. Contradiction! □

Lemma 5.4.10. *Let inh_0 be a State. Let inheritance function inh be an arbitrary extension of function inh_0 on a subset of $\mathcal{C}$.*
A) For every class $K \in dom^{RO}_{inh_0}$ and every word $w_0 \in \mathcal{A}_0^* = \{decl, inh_0\}^*$ and analogous word $w \in \mathcal{A}^* = \{decl, inh\}^* : w_0(K) = w(K) \in dom^{RO}_{inh_0}$ or both sides are undefined.
B) For every class $K \in dom^{RO}_{inh_0}$ and for every type X :
If for every $1 \le i < length(X)$ $\quad Bind_{inh_0}(X|^i \text{ in } K) \in dom^{RO}_{inh_0}$ then for all $1 \le i < length(X)$

$$Bind_{inh_0}(X|^i \text{ in } K) = Bind_{inh}(X|^i \text{ in } K)$$

and either

$$Bind_{inh_0}(X \text{ in } K) = Bind_{inh}(X \text{ in } K) \in \mathcal{C}^O$$

or both sides are undefined.

Proof. Proof of A)
Two cases are to be discussed: A1) $w_0(K) = K_n \in dom^{RO}_{inh_0}$ resp. A2) $w_0(K)$ is undefined.
A1) Because $inh_0 \subseteq inh$
we have $w(K) = K_n$ as well.
A2) Let $K_n \in dom^{RO}_{inh_0}$ be the final class in the chain of classes
$K = K_0, K_1, \cdots, K_n \in dom^{RO}_{inh_0}$ with $n < m$ which $w_0 = inh_{01} \cdots id_{0m}$ and K are inducing, $id_{0i} \in \mathcal{A}_0$. Then $id_{0,n+1}(K_n)$ is undefined. K_n is *Object* or *Root* because inh_0 is a State. Because $inh_0 \subseteq inh$ and $inh(Root)$ and $inh(Object)$ are undefined we have for the analogous word $w = id_1 \cdots id_m$,
$id_i \in \mathcal{A}$: Either $K_n = Root$ and $id_{0,n+1} = id_{n+1} = decl$ or $K_n \in \{Root,$
$Object\}$ and $id_{o,n+1} = inh_0, id_{n+1} = inh$. So $w(K)$ is undefined.
Proof of B)
(Base of induction $length(X) = 1$)
Due to A) the $\mathcal{A}_0$- resp. $\mathcal{A}$-chains of classes starting in K coincide. So Definition 5.4.1 does not differentiate between inh_0 and inh which the definition is based on.
(Induction step $length(X) > 1$)
Let for every $1 \le i < length(X)$

$$Bind_{inh_0}(X|^i \text{ in } K) \in dom^{RO}_{inh_0} \quad (\star).$$

Due to induction hypothesis and assumption $(\star)$ we have for all $1 \le i < length(X) - 1$

$$Bind_{inh_0}(X|^i \text{ in } K) = Bind_{inh}(X|^i \text{ in } K) \in dom^{RO}_{inh_0}$$

and

$$Bind_{inh_0}(X|^{length(X)-1} \text{ in } K) = Bind_{inh}(X|^{length(X)-1} \text{ in } K) \in dom^{O}_{inh_0}.$$

So we find an analogous situation as in the induction base and Definition 5.4.1 does not differentiate between inh_0 and inh. □

Lemma 5.4.11. *If inh is a State then the inheritance function*
$inh' = BDfl'(inh)$ is an extension of inh.

Proof. Let $A \in dom_{inh}$. Then $A \ne Root,\ A \ne Object,\ decl(A) \in$
$dom^{R}_{inh}, inh(A) \in dom^{O}_{inh}, inh(A) = Bind_{inh}(ext(A) \text{ in } decl(A))$ because inh is a State. So $\alpha^{B}_{inh}(A)$ is holding and $inh'(A) = Bind_{inh}(ext(A)$ *in*
$decl(A))$ by definition. So $inh'(A) = inh(A)$, i.e. inh' is an extension of inh. □

Remark 5.4.12. *Let inh be a State and* $A' \in \mathcal{C} \backslash dom_{inh}$ *with* $decl(A') \in dom^R_{inh}$ *(i.e.* A' *is a so called candidate) and* $Bind_{inh}(ext(A')\ in\ decl(A')) \in dom^O_{inh}$ *(i.e.* A' *is a so called generating candidate). Then let us denote the extension*

$inh\ \cup\ \{\langle A', Bind_{inh}(ext(A')\ in\ decl(A'))\rangle\}$

of inh by $inh^{A'}$*.*

Lemma 5.4.13. *If inh is a State then* $inh' = BDfl'(inh)$ *is also a State.*

Proof. Let $A \in dom_{inh'}$. We have to show that $inh'(A) \in dom^O_{inh'}$ and $decl(A) \in dom^R_{inh'}$ and that $inh'(A) = Bind_{inh'}(ext(A)\ in\ decl(A))$ is holding. We have two subcases
A) $A \in dom_{inh}$ and
B) $A \in dom_{inh'} \setminus dom_{inh}$.
Subcase A) is straightforward by help of Lemmas 5.4.9, 5.4.10, 5.4.11.
Proof of the subcase B): Because $inh'(A)$ is defined, $\alpha^B_{inh}(A)$ 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 $decl(A) \in dom^R_{inh}$, inh' is an extension of inh and $Bind_{inh}(ext(A)\ in\ decl(A)) \in dom^O_{inh}$ then we have due to Lemma 5.4.9 and Lemma 5.4.10 B)

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

The left side is exactly $inh'(A)$ by definition of $BDfl'$. □

Remark 5.4.14. *on direct and indirect successors of States:*
If in this proof of Lemma 5.4.13 we replace inh' *by* $inh^{A'}$ *then we have a proof for:* $inh^{A'}$ *is a State. We call* $inh^{A'}$ *a direct successor State of inh and write* $inh \prec^{DS} inh^{A'}$ *with the transitive closure* $\prec^S$ *of* $\prec^{DS}$ *which is an irreflexive partial order in the set of States* $\mathcal{C} \overset{State}{\longrightarrow} \mathcal{C}^O$. □

Proof. of Theorem 5.4.8, on monotonicity of $BDfl'$:
Let $inh_1 \subseteq inh_2$ be two States and $BDfl'(inh_1)(A) = inh'_1(A) = M$ be defined. We claim $BDfl'(inh_2)(A) = inh'_2(A) = M$.
Due to definition of $BDfl'$ we have that

$\alpha_{inh_1}(A) \wedge\ M = Bind_{inh_1}(ext(A)\ in\ decl(A))$

is holding.
Case 1: $A \in dom_{inh_1}$. Then $A \in dom_{inh_2}$ and $M = inh'_1(A) = inh_1(A) = inh_2(A) = inh'_2(A)$.
Case 2: $A \in dom_{inh'_1} \setminus dom_{inh_1}$. Then $M = Bind_{inh_1}(ext(A)\ in\ decl(A)) \in dom^O_{inh_1}$ Lemmas 5.4.9 and Lemma 5.4.10 B) 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 5.4.15. *on modular confluence:*
The relation $\prec^{DS}$ *is modularly confluent, i.e. if* $inh \prec^{DS} inh^A$, $inh \prec^{DS} inh^{A'}$ *and* $inh^A \neq inh^{A'}$ *then there is a common direct successor* $\widetilde{inh}$ *with* $inh^A \prec^{DS} \widetilde{inh}$ *and* $inh^{A'} \prec^{DS} \widetilde{inh}$*, especially* $\widetilde{inh} = inh^{AA'} = inh^{A'A}$ *(due to an easy calculation using Lemma 13). If a* $\prec^{DS}$*-chain*

$inh_0 \prec^{DS} inh_1 \prec^{DS} inh_2 \prec^{DS} \cdots \prec^{DS} inh_n$

ends up in a maximal inh_n *then* inh_n *is uniquely determined by* inh_0*. Every State* inh_0 *has such a uniquely determined maximal successor State* inh_0^{max}*. Obviously*

$BDfl'(inh) = inh \cup \{\widetilde{inh} : inh \prec^{DS} \widetilde{inh}\}$

is holding. Therefore a State inh is maximal w.r.t. $\prec^S$ *if and only if inh is a fixed point of*

$BDfl'$. The maximal successor State $inh_{\bot}^{max}$ is the least fixed point of $BDfl'$, obviously. If inh has no cycle then inh^A has none as well since $A \notin dom_{inh}^{O}$. If Dep_{inh} has no cycle then so it is for Dep_{inh^A}, because we may easily deduce by Lemmas 5.4.9 and 5.4.10 B)

$$Dep_{inh^A} = Dep_{inh} \cup \{\langle A, Bind_{inh}(ext(A) \mid^i \ in\ decl(A))\rangle : 1 \leq i \leq length(ext(A))\}$$

where $A \in \mathcal{C} \setminus dom_{inh}$ is the generating candidate for inh^A. □

Theorem 5.4.8 allows to apply the fixed point theorem: $BDfl'$ in

$$(\mathcal{C} \xrightarrow{state} \mathcal{C}^{\mathcal{O}}) \xrightarrow{tot,cont} (\mathcal{C} \xrightarrow{state} \mathcal{C}^{\mathcal{O}})$$

has exactly one least fixed point ($\kappa = card(\mathcal{C})$)

$$\mu BDfl' = \bigcup_{i \in Nat_0} BDfl'^{\,i}(inh_{\bot}) = BDfl'^{\,\kappa}(inh_{\bot})$$

which is

$$= \bigcup_{inh_{\bot} \prec^S inh} inh \ = \ inh_{\bot}^{max}.$$

Theorem 5.4.16. *If the given Java-program is Well-Formed then the function $Bind_{inh_{B0}}$ is satisfying all six rules of IPET. The rules are the same as in Theorem 5.3.2, only inh_0 , bind and $bind_{inh_0}$ are to be replaced by inh_{B0}, Bind and $Bind_{inh_{B0}}$ respectively.*

Proof. The verification of the rules I and II is the same as earlier in the proof of the theorem 5.3.2.
III. (ET-SimpEncl)
Consider a user declared class P. From the first premise: $Bind_{inh_{B0}}(D \text{ in} P) = T$ we have that there exist the word w_0 - the least word such that $w_0(P).D = T$. From the definition of *Bind*

$$Bind_{inh_{B0}}(ext(P.C).D \text{ in } P) = Bind_{inh_{B0}}(D \text{ in } Bind_{inh_{B0}}(ext(P.C) \text{ in } P).$$

Since inh_{B0} satisfies the property I_1 we can simplify the right-hand side of the equation to $Bind_{inh_{B0}}(D \text{ in } inh_{B0}(P.C))$.
The third premise reads: $Bind_{inh_{B0}}(ext(P.C).D \text{ in } P)$ is undefined. From it we have that for every word w'' of the form $w'^\frown inh$ the value of the expression $(w'^\frown inh)(P.C).D$ is undefined. From the second premise we have: the value of $\lambda(P.C).D$ is undefined. Now consider $Bind_{inh_{B0}}(D \text{ in } P.C)$. Remark that the equality $(w_0^\frown decl)(P.C).D = T$ holds. Making use of the observations based on the second and third premises we conclude that the word $w_0^\frown decl$ is the least word w such that the expression $w(P.C).D$ is defined. Hence, $Bind_{inh_{B0}}(D \text{ in } P.C) = T$. If $P.C = Object$ then $P = Root$ and $P.D = T$. The conclusion is holding also due to definition of $Bind_{inh_{B0}}$. If $P = Root$ then the conclusion is holding trivially.
IV. (ET-SimpSup)
The equality $inh_{B0}(P.C) = Bind_{inh_{B0}}(ext(P.C)\ in\ P)$ is valid if $P.C$ is a user declared class because condition I_{B1} is holding. Due to definition of $Bind_{inh_{B0}}$ we have $T = Bind_{inh_{B0}}(ext(P).D\ in\ P) = Bind_{inh_{B0}}(D\ in\ inh_{B0}(P))$ and T is $\mu w(inh_{B0}(P)).D$. Because $P.C.D$ is undefined P is not repeated on the path $\mu w\ inh_{B0}$ from P to $decl(T)$. So $\mu w\ inh_{B0}$ is the least path, denoted $\mu\tilde{w}$, without repeated classes such that $\mu\tilde{w}(P).D = T = Bind_{inh_{B0}}(D\ in\ P)$. If $P = Object$ or $= Root$ then the conclusion is holding trivially.

V. (ET-Long)
The conclusion is holding due to definition of $Bind_{inh_{B0}}$.
VI. (ET.LongSup)
The equality $inh_{B0}(T) = Bind_{inh_{B0}}(ext(T)\ in\ P')$ is valid if T is a user declared class because condition I_{B1} is holding. Due to definition of $Bind_{inh_{B0}}$ we have U is $\mu w(inh_{B0}(T)).C$ and $U = Bind_{inh_{B0}}(ext(T).C$ in $P') = Bind_{inh_{B0}}(C\ in\ inh_{B0}(T))$. Because $T.C$ is undefined T is not repeated on the path $\mu w\ inh_{B0}$ from T to $decl(U)$. So $\mu w\ inh_{B0}$ is the least path, denoted $\mu\tilde{w}$, without repeated classes such that $\mu\tilde{w}(T).C = U = Bind_{inh_{B0}}(C\ in\ T) = Bind_{inh_{B0}}(X.C\ in\ P)$.
If $P = Object$ or $P = Root$ then the conclusion is holding trivially. □
We may observe that the model $Bind_{inh_{B0}}$ may be calculated by an algorithm. Hence we have two different algorithms to construct the models of IPET calculus. Now we see that the statement "a straightforward algorithm ...[11]p.34" is not justified at all. First of all the authors of [11] gave no description of the algorithm. Second, there exist at least two algorithms. The cited statement does not answer to the question which of algorithms is the proper one?

5.5 The dilemma with IPET's rule III. (ET-SimpEncl)

The dilemma with IPET does not end itself on that IPET allows at least two different binding functions $bind_{inh_0}$ resp. $Bind_{inh_{B0}}$ which satisfy all six rules of IPET and which yield two different notions of well-formedness of (the structure of) a Java-program. If there were a clear criterion for the calculus how to elect the distinguished inheritance function resp. binding function everything would be fine. Let us remark that, in case all premises and conclusions are positive logical formulas then the intersection of any two satisfying functions is satisfying as well. Moreover, the distinguished function can be gained in a constructive manner by successive top-down applications of the rules.
We have already seen that rule III. (ET-SimpEncl) has a premise which is a negative, and a metatheoretic formula such that clear application of the rule is a great problem. It is even so that there is a program, namely Example 5.4.4, which is well-formed w.r.t. $bind_{inh_0}$ and $Bind_{inh_{B0}}$, but the intersection function $bind_{inh_0} \cap Bind_{inh_{B0}}$ does not satisfy rule III. (ET-SimpEncl), especially is different from $bind_{inh_0}$ which Java Language Specification JLS [7] has prescribed to be the appropriate binding function.

Lemma 5.5.1. *The intersection of the two models of IPET calculus need not to be a model of IPET.*

Proof. The proof of the lemma consists in exhibiting a counter-example program of the previous section.
Example 1 continued:
We have

$$inh_0(B) = inh_{B0}(B) = A$$
$$inh_0(B\$D) = inh_{B0}(B\$D) = A\$C$$
$$inh_0(B\$D\$F) = B\$E$$
$$inh_{B0}(B\$D\$F) = A\$E \neq B\$E \quad !$$

Let us calculate the binding functions:

$$bind_{inh_0}(\mathrm{A}\ in\ Root) = Bind_{inh_{B0}}(\mathrm{A}\ in\ Root) = A$$
$$bind_{inh_0}(\mathrm{C}\ in\ B) = Bind_{inh_{B0}}(\mathrm{C}\ in\ B) = A\$C$$

Now $bind_{inh_0}(E\ in\ B\$D) = B\$E$

because path *decl* from $B\$D$ to B is lexicographically less than path
inh⌢decl from $B\$D$ to A

But $Bind_{inh_{B0}}(\text{E } in\ B\$D) = A\$E \neq B\$E$!

because path *decl⌢inh* from $B\$D$ to A is lexicographically less than
path *decl* from $B\$D$ to B.

Let us define the intersection relation

$$Int \stackrel{df}{=} bind_{inh_0} \cap Bind_{inh_{B0}}.$$

We shall prove that Int relation does not satisfy rule III (ET-SimpEncl). From the facts gathered till now one deduce (using the rule III) that the value of $Int(E\ in\ B\$D)$ should be $B\$E$.

However from the definition of Int we have

$B = decl(B\$D$

$B\$D.E$ is undefined

$Int(EinB) = B\$E$

$Int(C.E\ in\ B)$ is undefined

because

$bind_{inh_0}(C.E\ in\ B)$

$= attrbind_{inh_0}(E\ in\ bind_{inh_0}(C\ in\ B))$

$= attrbind_{inh_0}(E\ in\ A\$C)$ is undefined

and

$Bind_{inh_{B0}}(C.E\ in\ B)$

$= Bind_{inh_{B0}}(E\ in\ Bind_{inh_{B0}}(C\ in\ B))$

$= AttrBind_{inh_{B0}}(E\ in\ A\$C)$

$= A\$E.$

But

$bind_{inh_0}(E\ in\ B\$D) = B\$E$

$Bind_{inh_{B0}}(E\ in\ B\$D) = A\$E$

hence

$Int(E\ in\ B\$D)$ is undefined.

Hence Int is not a model of IPET. □

In the previous sections we have unexpectedly seen two different models of IPET calculus. Therefore the IPET can not be treated as a definition of function of binding. Now, the immediate instinct to enrich the IPET with a (*metatheoretic*!) clause: "*choose the least of all IPET's models*" leads to nowhere.

5.6 What was left open by Igarashi and Pierce

The identification of declarative occurrence T of a class that is bound to an applicative occurrence of a (class) type X within a class P is basic for the understanding how a program works. The paper [11] offers the IPET calculus for deducing the values of the function $bind(X \text{ in } P) = T$, in original paper it is written $P \vdash X \Rightarrow T$. It turned out the formal system of IPET has many models, hence, the system does not define the binding function.

The discussion of this chapter shows how important is to state a few questions known already in metamathematics:

1. (*determinacy or consistency*) It is obvious that a formal system may allow to prove a sentence on many alternative ways. However, a sound system may not allow to deduce mutually negating answers. In this case the question should be: *is it true*

that for every class P and for every type X if IPET calculus allows to deduce two triplets $P \vdash X \Rightarrow T$ and $P \vdash X \Rightarrow U$ then $T = U$? We should be sure that the relation $P \vdash X \Rightarrow T$ is a function, that binds an applicative occurrence of the type X inside the class P to the declaration T of a class.

2. (*categoricity or completeness*) How many models has a proposed formal system? In our case the question is *are there different functions bind that are models of the IPET calculus?* The positive answer tells us that something important escaped our attention.

3. (*repairing an incomplete system*)If there are several models, one should try to repair the formal specification by adding either more axioms and inference rules (this way, we believe, is the correct one) or by adding some metatheoretic rule like, *for example, among all possible models choose the least one.* Or better, among all possible models choose the one calculated by certain algorithm.

These questions were not addressed in the paper [11].

Still there are open questions (to be answered in forthcoming papers):

- is it possible to repair the IPET calculus and present a formal system which will enable to deduce the values of the function $bind_{inh_0}$?
- how many models the calculus IPET admits? The eventual answer may interest the designers of programming languages.

Chapter 6

Static and runtime binding

In this chapter we shall discuss the problem of determining the meaning of identifiers during a compilation and later an execution of a program. In the preceding two chapters we discussed the problem of identifying the direct superclass of a given class. We gave the algorithm for computing function inh. This algorithm uses another function bind. The latter algorithm finds application during the phase of compilation which checks whether a source code is well formed program. This phase of compilation is usually called "static semantic analysis". The application of function bind during compilation allows to simplify the access to variables during execution of program as it will be seen from the theorem 6.1.6.

6.1 Binding of variables

Variables are referenced in method bodies. Variables declared inside a method are considered to be local. All other references to variable identifiers should be bound in the environment of the method. This environment consist in the class which declares the method and all classes textually enclosing the declaring class. Without loss of generality we assume that all variables are of class types. We assume that for every class K there exist a function .v such that K.v returns the class M being the type of the variable v declared within the class K or remains undefined if the class does not declare a field variable named v. Since no class should declare two different variables with the same name K.v is really the function.

Definition 6.1.1. *An applied occurrence of a variable identifier v within the body of a method M will be called nonlocal iff the variable identifier v is not declared within the method M neither as a local variable nor as a formal parameter.*

The way of searching the environment of nonlocal variable v applied occurrence in method M body is isomorphic to a class identifier binding when they occurs within classes (cf. definition 2.1.2).

Definition 6.1.2. *Let K be a class. An applied occurrence of a variable identifier v in the class K is* bound *to a class L such that L is the type of the applied variable v occurrence.*

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

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

Definition 6.1.3. *The nonlocal applied occurrence of a variable identifier v with the body of a method M is bound to its type L iff its bound to the same type as applied variable occurrence in class K where K is the class declaring the method M.*

During runtime applied occurrences of variable identifiers should be bound to the corresponding locations in store. Each class together with its declared variable fields is the pattern used to allocate location for every variable declared in this class. During runtime any given class may be used many times to reserve new locations to declared field variables. All locations for field variables of a given class should be reserved at once and we call such collection of locations a data field of the class. As we will see in the next section data fields forms during the run time a structure

$$\mathcal{DF} = \langle Dfields, Decl, Inh \rangle$$

homomorphic to the compile time structure

$$\mathcal{CL} = \langle Classes, decl, inh \rangle$$

with the homomorphism $h : Dfields \longrightarrow Classes$ defined as follows
$h(\kappa) = K$ iff the data field κ was reserved for locations of class K field variables.

Since field variables have unique names within each class we can assume that for every data field κ there exist function .v such that κ.v returns the location of variable field v within data field κ iff such location exists

Definition 6.1.4. *Let κ be a data field of an arbitrary class. An applied occurrence of a variable identifier v in the data field κ is* bound *to a location λ such that λ is the value of the function bind computed in the following equality.*

$$\boxed{bind(v\ \ in\ \kappa) \stackrel{df}{=} (Inh^i Decl^j(\kappa)).v}$$

where the pair (j, i), $j \geq 0$, $i \geq 0$, is the least pair in the lexicographic order such that the class $(Inh^i Decl^j(K)).v$ is defined.

Definition 6.1.5. *The nonlocal applied occurrence of a variable identifier v while invocation φ of the method M is bound to its the location λ iff its bound to the same location as applied variable occurrence in class data field κ where κ idata field of the classs defined as the value of the expression "this" within the invocation φ*

Theorem 6.1.6. *Let κ be a data field of class K, v be applied variable identifier occurring in κ and in K and L be a class designated as type of v by the formula*
$(inh^{i_{stat}} decl^{j_{stat}}(K)).v = L$
from the definition 6.1.2 and λ be a location designated as the location of v by the formula
$(Inh^{i_{dyn}} Decl^{j_{dyn}}(\kappa)).v = \lambda$
from the definition 6.1.4
then $j_{stat} = j_{dyn}$ and $i_{stat} = i_{dyn}$

6.2 Instantiation of classes

6.2.1 Static typing of instantiation expression

Class instantiation is done by the expression `new`. The result of class instantiation is producing a class instance. A class instance consists of data fields for the instantiated class

and data fields for all its superclasses. The expression `new` may be invoked in one of two contexts
`e.new C(...)` or
`new X(...)` as top level expression
In both cases the first step is to determine the declaration of the class to be instantiated. Static semantics analysis requires the declaration of the instantiated class while in run time we are obliged to define the values of functions decl and Inh as well for data field of the instantiated class as for data fields of all its superclasses, which are to be reserved simultaneously.

Let we start with determination of the class to be instantiated. If the new expression is in the first form then the expression e should be well typed to some class K. Class identifier C is bound to the class L iff
$L = (inh^i(K)).C$
for some i and i the least number ≥ 0 such that the right hand side of the equality is defined.
For the second form we must first establish
$L = bind(X\ in\ K)$
where class K is the owner of the method instantiating class. Then we must check whether for some i and j
$decl(L) = inh^i decl^j(K)$

Lemma 6.2.1. *Let new X(...) be an class instantiation expression within a class K, and let length(X)=1 i.e. X=C where C is a class identifier. Then the type of the expression new X(...) is equal bind(C in K)*

The observation stated in the lemma above does not generalize. It is possible that if the type of the expression new X(...) in K is L then L $\neq$ bind(name(L) in K) as it can be seen from the following

Example 6.2.2. *Class instantiation.*

```
class B1{
    class A {}
    class B2 {
        class A {}
        void f() {
            new B1.A();
            new A();
        }
    }
}
```

Expression new B1.A() instantiates class B1$A of an identifier A, while expression new A() instantiates class B2$A.

6.2.2 How to generate class instance using class instance generator

Definition 6.2.3. *Let κ be a data field of class K. This data field may be used as class instance generator for any direct inner class of K.*

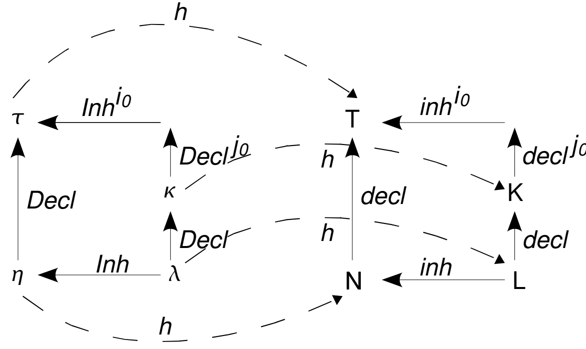


Figure 6.1: Finding a class instance generator τ for generation of class N instance η where $N = inh(L)$ and $decl(L) = K$ $\kappa = h(K)$ are known.

The aim of a class instance generator κ is to instantiate a direct inner class. Let it be the class L with the $name(L) = C$. Each class instance consists of data field of an instantiated class and data fields of its all superclasses. Each of the data fields should have defined its $Decl$ and Inh functions properly. So, the first task of class L generator is to generate an instance for direct superclass of the class L. Since it is done we obtain instance of superclass consisting of data field of all superclasses of L, each data field with functions $Decl$ and Inh already defined. Then we reserve new locations for data field λ of class L and define $Inh(\lambda) = \eta$ and $Decl(\lambda) = \kappa$ where η is data field of the direct superclass of the class L.

Hence it is only left to define how class instance generator instantiates the direct superclass of a class to be instantiated. In other words the only problem is how to find a direct superclass L generator when the class L generator is known as κ. We remain that it means that $Decl(L) = h(\kappa)$. In order to deal with this problem we have class L constructor of the form.
C(...){ e.super(...)} or
C(...){ super(...)}
where it is assumed that name(L) = C and $inh(L) = N$
In order to better understand the second form its worth to remain that the expression super should instantiate the superclass which is already known from the earlier stage of static semantics analysis as N. Let we treat the super() expression as a fictious class generator new N(), in a very specific form where the class identifier is changed by the class itself. Now we check if a formula
$decl(N) = inh^i decl^j(K)$ $(*)$
holds for any $i, j \geq 0$. This test may be performed during static semantics analysis and for well typed program at least one such formula should hold. However if we have more candidates we have a problem how to define class instance generator for N. In fact we choose, perhaps a little bit arbitrary, to set class N instance generator as data field τ of the class $T = h(\tau) = decl(N)$ where $\tau = Inh^{i_0} Decl^{j_0}(\kappa)$, such that and j_0 is the least integer such that the formula $(*)$ holds. Its worth to notice that for such j_0 i_0 is defined uniquely since there is no cycle in the graph of the inheritance function $inh(L)$

If the L constructor is in the first form then we assume, as in the previous case that the direct superclass of L is N and $decl(N) = T$. Then it is a static semantics analysis responsibility that the static type E of e is the subtype of T. During run time we must evaluate the expression e before class N is instantiated. The expression e is evaluated in

the dynamic context of κ instance of class K which is the class instance generator of class L. Let η designates the value of e. We assume the dynamic run time type correctness which means that the type of $h(\eta)$ is a subtype of E. In other words $Inh^j(h(\eta)) = inh^i(E) = T$ for some $j \geq i$. So $Inh^j(\eta) = \tau$ where $h(\tau) = T$. Then τ is the instance generator of superclass N.

6.2.3 How to find the proper class instance generator from the place of class instantiation.

Let we assume that the class instantiation expression new X(...) occurs at the time of the method M execution and at this time the value of the expression this within method M is υ. Let M be a method of a class K. Then $K = h(Inh^k(\upsilon))$ for some k. Then assume that static analysis has defined the class to be instantiated as L and j is the smallest integer such that $decl(L) = inh^i decl^j(K)$. Note that for such j the number i is defined uniquely while the existence of such j is guaranteed by static semantics analysis. Then the L class instance generator is $\tau = Inh^i Decl^j Inh^k(\upsilon)$

Now it remains to define class instance generator for the expression of the form e.new C(...). Let K be a static type of the expression e and C be bound to class L by static semantics analysis. It means that $decl(L) = inh^i(K)$ for an uniquely defined i. Then let $h(\nu) = E$ where ν is run time value of the expression e. So $K = inh^j(E)$ for some j due to run time type correctness. Then the class L instance generator is $\tau = Inh^i Inh^j(\nu)$

Chapter 7

Conclusions

The Java programmers and the compilers of Java must resolve the problem of which of possibly many classes B is extended by the class A extends B{...}. This problem is not trivial because Java admits inner classes. The problem becomes even more complicated, when we recall more general form of class declaration

$$class\ A\ extends\ B.C.D\ \{\ldots\}.$$

In the specification of Java language [7] one may find conditions which must be satisfied by the proper solution. This specyfication however doesn't help to find a proper solution. It also doesn't guarantee nor that a solution exists neither that the solution is unique.

The problem, treated informally in JSL[7] was was given an algebraic formulation in Chapter 2. Chapter 3 brings a non deterministic algorithm and the proof of its correctness. Next chapter presents a deterministic algorithm together with its analysis. Chapter 5 discusses earlier works on this problem in various Object Oriented languages. A special attention was given to the work of Igarashi & Pierce [11]. This paper contains a system of axioms and inference rules which were invented in order to give reduction semantics of the subset of Java. We proved that this work has a flaw.

There are at least two possible applications of the results of this dissertation. First, the algorithm determining the extended classes may and should be a part of any Java compiler. Second, the theoretical work made in the context of the problem may constitute a good foundation for the future research of the semantics of Java programming language. Chapter 6 may be treated as a first step of such research. The research which may lead to, among the others, an extension of the Eclipse environment and an axiomatic definition of the Java programming language.

Index

binding of identifiers, v, 1
- at compile time, **9**, 13, 29, 38, 41, 47, 49–51
- at run time, 50
- of classes, **9**, 13, 29, 38, 41, 47, 51
- of variables, 49, 50

class constructor, 52, **52**
class instance generator, **51**, 53
class instantiation, 50, 53

data field of the class, 50, **50**
dependency of classes, 3, **10**
- cycle of, 2, 10, 18, 19, 21–23, 27, 31, 42

ordered collection of sons, **23**

State, 42–44
static semantics analysis, 52, 53
stripped program, **7**, 8, 10

type, 2, 3, 7, **8**
- empty, 7, 8
- qualified, v, 7

Bibliography

[1] W. M. Bartol et al., *The Report on the Loglan'82 Programming Language*, PWN, 1984, Warszawa

[2] 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, Page* http://www.daimi.au.dk/~beta (as of 2001)

[3] W.M. Bartol, A. Kreczmar, A.I. Litwiniuk, H. Oktaba, *Semantics and Implementation of Prefixing on Many Levels*, in Proceedings Logics of Programs and Their Applications, A. Salwicki ed., LNCS 148, Springer, 1983, Berlin

[4] 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, 1984, Kiel

[5] G. Cioni, A. Salwicki,eds. *Object Oriented Programming*, Academic Press, 1987, London

[6] *Inner Class Specification*, http://java.sun.com/products/jdk/1.1/docs/guide/inner/classes (of 1997)

[7] J. Gosling, B. Joy, G. Steele, G. Bracha, *The Java Language Specification*, third edition, Addison-Wesley, 2005
$http://java.sun.com/docs/books/jls/$

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

[9] 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, http://www.iro.umontreal.ca/~simula (of 2002)

[10] 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 2635, Springer,209-235, 2004

[11] A. Igarashi, B. Pierce, *On inner classes*, Information and Computation **177** (2002), pp. 56-89

[12] B. Eickel, *Thinking in Java* Prentice Hall, fourth edition, 2005

[13] G.Mirkowska, A. Salwicki, *Algorithmic Logic*, PWN & D.Reidel, Warsaw & Dordrecht, 1987

[14] A. van Wijngaarden et al. (eds.). *Report on the Algorithmic Language ALGOL68.* Numerische Matthematic, 79-218, 1969

[15] H. Rasiowa, R. Sikorski, *Mathematics of metamathematics,* PWN Publ., Warsaw, 1963

---

[16] H.Langmaack, A.Salwicki, M.Warpechowski, *On correctness and completeness of an algorithm determining inherited classes and on uniqueness of solution* G.Lindemann et al. Proc. CS&P'2004 pp. 319-329

[17] H.Langmaack, A.Salwicki, M.Warpechowski, *On an algorithm determining direct superclasses in Java-like languages with inner classes – its correctness, completeness and uniqueness of solutions.* Information and Computation, **207**(2009) pp.389-410

[18] H.Langmaack, A.Salwicki, M.Warpechowski, *Some methodological remarks inspired by the paper "On inner classes" by Igarashi and Pierce* Proc. Conf. CS&P'2008 Gross Vaeter See, pp. 348-359

[19] H.Langmaack, A.Salwicki, M.Warpechowski, *On an deterministic algorithm identifying direct superclasses in Java* Fundamenta Informaticae **85**, pp.343-357

[20] H.Langmaack, A.Salwicki, M.Warpechowski, *On correctness and completeness of an algorithm determining inherited classes and on uniqueness of solutions* Proc. CS&P'2004 pp. 319-329

[21] A. Salwicki, M. Warpechowski, *Combining Inheritance and Nesting of Classes Together: Advantages and Problems*, in Proc.Conf. CS&P'2002, Berlin, Humboldt Universität, pp. 305-316

[22] A. Kreczmar, A. Salwicki, M. Warpechowski, *Loglan'88 - Report on the Programming Language*, LNCS 414, Springer, 1990, Berlin

[23] M. Krause, A. Kreczmar, H. Langmaack, M. Warpechowski, *Concatenation of Program Modules, an Algebraic Approach to the Semantics and Implementation Problems*, in Proceedings Computation Theory, A. Skowron ed., LNCS 208, pp.134-156, Springer, 1986, Berlin