Some methodological remarks inspired by the paper "On inner classes" by A. Igarashi and B. Pierce

Hans Langmaack

Institut für Informatik, Christian-Albrechts-Universität zu Kiel
Christian-Albrechts-Platz 4, D-24098 Kiel, Germany

Andrzej Salwicki

National Institute of Telecommunication, Warsaw
Szachowa 1, 04-894 Warszawa, Poland

Marek Warpechowski

Institute of Informatics, Warsaw University
Banacha 2, 02-092 Warszawa, Poland

Running title:
Some methodological remarks

Address for correspondence:
Andrzej Salwicki
salwicki@mimuw.edu.pl
National Institute of Telecommunication
Szachowa 1,
04-894 Warszawa, POLAND
tel. 00 48 22-512-83-62
fax 00 48 22-512-84-00

Abstract

In [IP02] an axiomatic approach towards the semantics of FJI, essentially a subset of the Java-programming language is presented. At a first glance the approach of reducing Java's semantics to that of FJI seems promising. We are going to show that many questions were left unanswered. It turns out that the theory how to elaborate or bind types and thus to determine direct superclasses proposed in [IP02] has different models. Therefore the suggestion that the formal system of [IP02] defines the (exactly one) semantics of Java is not justified. It is popular in informatics to propose a set of inference rules and to claim that a semantics is defined in this way. Sometimes such a system contains a rule with a premise which reads: *there is no proof of something*. One should note that this is a metatheoretic property. It seems strange to accept a metatheorem as a premise, especially if such a system does not offer any other inference rules which would enable a proof of the premise. We are going to study the system in [IP02] to display its features.

Key words: object oriented programming, semantics, inheritance, inner classes, direct superclass, static semantics analysis, static binding, derivation calculus, model, minimal resp. least model

1 Introduction

The Java-programming language is one of a few languages which allow inheritance and inner classes. The combination of these two features makes the language interesting for software engineers. On the other hand, to define its semantics is a challenge. In [IP02] Igarashi and Pierce presented an axiomatic approach towards the semantics of the language FJI (Featherweight Java with Inner classes, essentially a subset of Java) and they reduced Java's semantics to that of FJI. In this paper we shall analyze one important subgoal: how to identify the direct superclasses in view of results in [IP02]. We are going to demonstrate flaws in their approach. Our remarks may apply as well to a wider collection of papers. Namely inference rule systems are popular in informatics, but due to their more or less informal presentations they sometimes conceal the serious problem of having a metatheoretic property as a premise, especially if such a system does not offer any inference rules which would enable a proof of the premise.

A declaration of a class may contain the keyword **extends** followed by the type X naming the direct superclass. An example declaration may look like this: **class** A **extends** $B.C$ $\{ \ldots \}$. Now, since classes may be declared inside classes (and methods) it may happen that there are several classes named B resp. C in one program. Which of the classes named C is the direct superclass of class A? Which of the classes named B should be used in the process of identification of the direct superclass of class A? Notice, it may happen that no correct direct superclass exists, even if there are many candidates.

Subsection 5.2.1 of section 5 of [IP02] is devoted to the elaboration of types which shall make the identification of direct superclasses possible. Table Fig. 14 of paper [IP02, section 5.2.1] cites six inference rules. The authors define a calculus; we name it IPET-calculus and analyze it. The calculus' aim 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 a certain class. One may say the calculus is inconsistent.
(2) Moreover, there exist at least two different models[⋆)] of the calculus.
(3) Moreover, the models do not enjoy properties of this kind: the intersection of two models is a model; or there is a least model; or there is at most one minimal model. Therefore it is difficult to say what the meaning of the calculus is.

The authors of [IP02] 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. Their so called algorithm is not an algorithm. For it does not enjoy the termination property, c.f. [IP02, section 5.2.1]. Therefore we propose to call it a *method*. Furtheron the method may lead towards different answers. We shall show that the method can be specified in at least two different manners. The IPET-calculus may be used to define the inheritance function *inh* from classes to classes. We can go 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 every model can be constructed by a corresponding algorithm. Hence it is necessary to add some hint of metatheoretical nature. Frequently, 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 and there are at least two different minimal models.

It seems that the source of the problems is in admitting a special 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 $P \vdash X.D \Rightarrow T$ which says: Type $X.D$ in (resp. directly enclosed

⋆ The word model is used informally here

by the body of) class occurrence P elaborates (is bound) to class occurrence T. One remedy would be to eliminate the rule and to replace it by some rules that do not introduce metatheoretic premises and such that the premises are positive formulas. Another approach would consist 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 [IP02].

We are stressing the fact that there exist several papers of various authors where the reader discovers a metatheorem as a premise of an inference rule. Therefore our remarks are of general character. A further most important motivation to write an article like the present one is: Since 1996 Java [GJS96] has turned out to be a most successful and very widely spread programming language used in academic institutions and above all in industrial practice. So computer science is severely obliged to prevent proliferation of errors and erroneous beliefs which are due to unclear interpretations of Java's language specification.

Igarashi's and Pierce's paper [IP02] is, in a sense, in the same tradition where software researchers have tried to understand and implement nested program structures. Implementation engineering witnessed serious set-backs because even renowned computer scientists had too naive views at or were too wishfully thinking about static binding and because practitioners were following blindly.

In 1960 Dijkstra [Dij60] has claimed that the so called static pointer in an Algol-runtime stack always points (has to point) to the most recently stacked activation record of the not yet completed activation of the first procedure body that lexicographically (textually) encloses the body of the procedure called. Dijkstra's advice does not correctly implement all Algol60-programs (a first counterexample is in [GHL67]), but only those programs which are so called "most recent"-correct [McG72] or satisfy the

so called “most recent”-property [Kan74] and therefore have so called regular call trees [Old81].

In 1965 McCarthy et al. [McC+65] presented Lisp-interpreters written in Lisp itself in order to define the semantics of this language. The interpreters had errors so that all Lisp-programs were treated in a way as if their call trees were regular. The authors did not intend that program property because Lisp was designed to be an extension of the applied lambda calculus where there are call trees which are not regular in general.

Papers like [Dij60,McC+65] have caused severe set-backs to good language implementation engineering for about two decades. The evoked discrepancies in language implementations have led to regrettable decisions of designers: Procedures and functions as parameters and thus formal procedure and function calls were disallowed in Ada [Ich80]. Languages like Smalltalk [GoRo89] and original Java 1996 [GJS96] allow only flat programs with non-nested classes and methods and so avoide irregular call trees by force. But embedded software design which leading computer scientists are advocating, see [Bjo09], has to comply with static scoping, especially because static scoping is constitutive for accompanying predicate logic formulas. Programs do not become better intelligible by artifcially enforced restriction to programs with regular call trees or by dynamic binding of names instead of static binding. Quite contrary, programs in Lisp and similar languages were more difficult to analyze. For dynamic binding may change the meaning of one and the same identier during a programs execution what is hard to pursue mentally.

In order to move Java into a direction where object orientation is in concordance with nesting of program structures, static scoping and embedded software design and thus to follow the lines of Simula67 [DaNy67], Loglan82/88 [Bar+82,KSW88] and Beta [MMPN93] the authors of Java have created their new Java Language Specification in 2000 [GJSB00]. Igarashi and Pierce supported this development by their article [IP02] and former con-

tributions.

The structure of our paper is as follows: Section 2 presents the calculus of Igarashi and Pierce. Examples are given showing that the calculus is inconsistent. We are giving a first simple remedy to it. In Section 3 we translate the inference rules of the IPET-calculus in such a way that the phrase "*the meaning of type X in environment P is class T*" is now expressed by the formula $bind(X \text{ in } P) = T$ and we show that the function $bind_{inh_0}$ calculated by the algorithm LSWA of [LSW09] is a model of the IPET-calculus. Next section shows that there is another concept $Bind_{inh_{B0}}$ of *binding* function and that $Bind_{inh_{B0}}$ is modeling the IPET-calculus uniformly for all programs as well. In the fifth section we show that the intersection of two models needs not be a model. So the hope is shrinking that by adding the – metatheoretic – phrase "*take the least model of the models of the IPET-calculus*" the task of identifying the direct superclasses of classes of a Java-program can be completed by the IPET-calculus. Indeed: Investigations on minimal models assure that there are at least two minimal models and there is no least model. Section 6 shows how to define IPET in a way conform to logics.

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

Igarashi and Pierce [IP02, 5.2.1] 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) *class declaration occurrence* `T`. In other words: the meaning of type `X` in class `P` is class `T`. For clarification: we have to differentiate between a syntactical entity and its occurrences, see the Algol68-report [Wij+68], because one and the same class declaration text (class

for short) may occur several times at different places 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`, $\mathtt{C}_{n-1},\cdots,\mathtt{C}_1$ are the names of the successive class occurrences which enclose class occurrence `P` and $\mathtt{C}_1$ names a top-level class. To understand this phenomenon one should notice that the classes of a program form a tree. The root of the tree is a fictitious class *Root* which directly 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 node n are the sons of node n. Therefore we are entitled to identify an occurrence of a class declaration and the absolute path of it. FJI requires that the extends clause has an extends type which is the absolute path of the denoted class occurrence whereas Java allows extends types which are not necssarily absolute paths. 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 (and implicitly all other class occurrences) of the Java-program and which has no name nor extends clause. The authors of [IP02] 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 no extends clause also. Without loss of generality we may 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 a premise of the form $CT(P.C) =$ **class** `C` **extends** `X` $\{\cdots\}$. In this way the authors Igarashi and Pierce express the fact that user declared class $P.C$ extends type X, i.e. the class which is the meaning of type X in that place

where the declaration of class $P.C$ occurs. Formulas of the form $P.C \in Dom(CT)$ mean: the program contains the class named C in its directly enclosing class which is identified with 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 Table 1 we present Igarashi's and Pierce's calculus IPET for elaboration of types. Below we collect some observations and comments.

Table 1
Igarashi's & Pierce's rules of elaboration

I. (ET-Object)	$P \vdash \texttt{Object} \Rightarrow \mathit{Object}$
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)	$\dfrac{P.C.D \notin dom(CT) \quad CT(P.C) = \textbf{class } \texttt{C } \textbf{extends } \texttt{X}\{\cdots\} \quad P \vdash X.D \Rightarrow T}{P.C \vdash D \Rightarrow T}$
V. (ET-Long)	$\dfrac{P \vdash X \Rightarrow T \qquad T.C \in dom(CT)}{P \vdash X.C \Rightarrow T.C}$
VI. (ET-LongSup)	$\dfrac{P \vdash X \Rightarrow P'.D \qquad P'.D.C \notin dom(CT) \qquad CT(P'.D) = \textbf{class } \texttt{D } \textbf{extends } \texttt{Y}\{\cdots\} \qquad P' \vdash Y.C \Rightarrow U}{P \vdash X.C \Rightarrow U}$

(1) The system IPET is inconsistent! Consider a Java-program with a user declared class Object named `Object`. From axiom I. (ET-Object) one obtains that the meaning of the name `Object` is *Object* (or $\star$.`Object`). From rule II. (ET-InCT) one obtains P.`Object` where P is the class containing the user declared class Object named `Object`. Imagine what will happen when there are several user declared classes Object in a program.

Remedy: The authors of the present paper are aware of this and require that a program cannot contain a user declared class named `Object`. Note that the remedy is inconsistent with the Java Lanaguage Specification [GJSB05] which allows to declare many user declared classes named `Object`. (A good reformulation of IPET calculus should liberate from this restriction).

(2) 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 proof? What instructions are given to her/him making the task possible to recognize the impossibility of any proof?

(4) A reader may hesitate to perceive what the following sentence is meaning: "*A straightforward elaboration algorithm obtained by reading the rules in a bottom-up manner might diverge.*" [IP02, 5.2.1, p.82]. 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 [RS63] 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 follows: 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 which 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 of [IP02] are aware that proof construction is not always possible. They make evident that the algorithm we guessed may loop without exit [IP02, 5.2.1, p.82].

(6) In fact the task of type elaboration is divided in two subtasks: a) to find whether 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 IPET does not help detecting the possible errors in typing.

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

(8) The next question arises: 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 Section 5.

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

In [LSW08,LSW09] Langmaack, Salwicki and Warpechowski studied structures of Java's programs and developed the family of binding functions. Let π be a Java program. By $\mathcal{C}^\pi$ we denote the

set of all user declared class occurrences of program π. By $\mathcal{CT}^\pi$ we denote the set of class types and by $\mathcal{SCT}^\pi$ the set of simple class types of the program π. For every program π of Java, $bind^\pi_{inh}$ is a function

$$bind^\pi_{inh} : \mathcal{CT}^\pi \times (\mathcal{C}^\pi \cup \{Root, Object\}) \to (\mathcal{C}^\pi \cup \{Object, null\})$$

which for given class type X and class occurrence P determines class occurrence T - the meaning of class type occurrence X directly enclosed by the class occurrence P. Our notation exhibits the fact that binding function *bind* depends on a given inheritance (i.e. direct superclassing) functions *inh*. The notation

$$bind^\pi_{inh}(X\ in\ P) = T \quad .$$

reads:*Let P be a class occurrence of program π, X a class type. Class T is the meaning of class type X inside the class P* - where T is a class occurrence or null. The value null signals that no correct meaning of class type X may be found inside the class occurrence P. (In the sequel we shall omit the superscript π as always at most one program will be discussed.)

Above that the authors developed an algorithm LSWA which uniformly with respect to programs determines superclassing function inh_0 which is the least fixed point of the continuous functional $Bdfl'(inh)$ see [LSW09]

$$inh_0 = Bdfl'(inh_0) = \mu Bdfl'$$

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 (their set is denoted $\mathcal{C}$) $inh_0(P)$ 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 [IP02] and of [LSW08,LSW09] ought

to agree the ternary relation

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

should comply with 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 [LSW08,LSW09] what is yielding the formulation of Definition 1.
Let π be a syntactically correct program in Java. We recall that the structure of classes of the program has the following properties:

- the set of classes considered with the relation *decl* such that *class decl(B) is the least class enclosing class B* is a tree,
- each class has a name,
- each class has an extends clause which is a simple or qualified type, i.e. a finite sequence of class names.

Here the empty extends clause is disallowed; the classes of programs considered in [LSW09] allow empty extends clauses. Names of classes are identifiers.

Definition 1 *The calculus BIPET is defined by the rules of Table 2.*

The calculus is not a formalized theory for it lacks a precise definition of the language used and the logical tools used. Therefore we can not use the term *model* of theory. Instead we shall say that a function f *complies with the rules* of BIPET if whenever the premises of one of inference rules of BIPET calculus are satisfied by the function f, the conclusion of the rule is satisfied too.

Table 2
Rules of calculus IPET are translated into calculus BIPET

I. (BET-Object)	$bind(\texttt{Object}\text{ in }P) = Object$
II. (BET - InCT)	$\dfrac{\text{class } P \text{ has a direct inner class named } C}{bind(C \text{ in } P) = P.C}$
III. (BET-SimpEncl)	$\dfrac{\begin{array}{l}bind(D \text{ in } P) = T \\ \text{class } P.C \text{ has no direct inner class named } D \\ bind(ext(P.C).D \text{ in } P) = \text{null}\end{array}}{bind(D \text{ in } P.C) = T}$
IV. (BET-SimpSup)	$\dfrac{\begin{array}{l}bind(ext(P.C).D \text{ in } P) = T \\ \text{class } P.C \text{ has no direct inner class named } D\end{array}}{bind(D \text{ in } P.C) = T}$
V. (BET-Long)	$\dfrac{\begin{array}{l}bind(X \text{ in } P) = T \\ \text{class } T \text{ has a direct inner class named } C\end{array}}{bind(X.C \text{ in } P) = T.C}$
VI. (BET-LongSup)	$\dfrac{\begin{array}{l}bind(X \text{ in } P) = P'.D \\ \text{class } P'.D \text{ has no direct inner class named } C \\ bind(ext(P'.D).C \text{ in } P') = U\end{array}}{bind(X.C \text{ in } P) = U}$
	Variables P, P' range over $\mathcal{C}^{RO}$, T, U over $\mathcal{C}^{O}$, X over $\mathcal{CT}$ and C, D over simple types

Theorem 2 *If the given Java-program π is well-formed then the function $bind^{\pi}_{inh_0}$ complies with all six rules of the BIPET-(and hence with the 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 (direct superclassing) functions *inh* which

parameterize $bind_{inh}$.
II. (BET-InCT)
If class P contains a direct inner class named C, i.e. $P.C$ is defined, $P.C \in Dom(CT)$, then the conclusion *the meaning of name C in class P is class $P.C$* is holding independently of all possible inheritance functions *inh*.
III. (BET-SimpEncl)
Let $P.C$ be a user declared class. From the first premise $bind_{inh_0}(D \text{ in } P) = T$ we have that there exist natural numbers $i \geq 0,\ j \geq 0$ such that $T = inh_0^i(decl^j(P)).D$ where the pair $\langle j, i\rangle$ is the least in the lexicographic order such that the right hand side expression is defined. The third premise "$bind_{inh_0}(ext(P.C).D \text{ in } P) = \text{null}$" 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

$$inh_0^l(inh_0(P.C)).D \text{ is undefined,} \tag{1}$$

because function inh_0 enjoys property

$$I_1' :\ inh_0(P.C) = bind_{inh_0}(ext(P.C) \text{ in } P)$$

or both sides are undefined. 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 expression $inh_0^l(decl^k(P.C)).D$ has a defined value and that the pair $\langle k, l\rangle$ precedes or is equal the pair $\langle j+1, i\rangle$. From the second premise and the property (1) we know that $k \neq 0$. In other words: the path $inh_0^l(decl^k(P.C)).D$ goes from class $P.C$ through 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 $l = i$.
IV. (BET-SimpSup)
Let $P.C$ be a user declared class. From the first premise $bind_{inh_0}(ext(P.C).D \text{ in } P) = T$ we have that there exists a natural number $i \geq 0$ such that the right hand side of $T = inh_0^i(bind_{inh_0}(ext(P.C) \text{ in } P)).D$ is defined. Since function inh_0 enjoys property I_1' we know that the expression $inh_0^i(inh_0(P.C)).D$ has a value T. Hence

$T = inh^{i+1}(decl^0(P.C)).D = bind_{inh_0}(D \text{ in } P.C)$ because 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 critical candidate pair $\langle 0, 0\rangle$ which is less than those pairs considered is excluded by the second premise.
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 $bind_{inh_0}(ext(P'.D).C \text{ in } P') = U$. This means that there exists a natural number $l \geq 0$ such that $inh_0^l(bind_{inh_0}(ext(P'.D) \text{ in } P')).C = U$ and l is the least one such 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 is the least one such that the left hand side is defined. The satisfaction of rule VI. follows from the definition of function $bind_{inh_0}$ [LSW09] and from the first premise. ■
Reading carefully the proof we observe the following

Fact 1 (A strengthening of Theorem 2) *The function $bind_{inh}$ complies with the six rules of BIPET even if only the inheritance condition I_1 for inh_0 holds, but not necessarily the non-cycling condition I_2 for dep_{inh_0}. Moreover, the condition I_1 may be weakened towards I_1': For all classes $P \in \mathcal{C} : inh_0(P) = bind_{inh_0}(ext(P) \text{ in } decl(P))$ or both sides are undefined, i.e. if inh_0 is a fixed point of the so called natural functional Bdfl [LSW09].*

Hence if I_1' holds for inh_0 then the binding function $bind_{inh_0}$ satisfies all six rules of IPET, i.e. is a *model* of IPET, especially if the given Java-program is well-formed, i.e. inh_0 satisfies I_1 and I_2 [GJSB05,LSW04,LSW09]. Our next question is: is IPET defining a unique model? Are there other binding functions which satisfy all rules of IPET? Is there a distinguished model of IPET? In what sense does IPET define a distinguished model? Or, perhaps, is there no reasonable way to distinguish a good inheri-

tance function? We have already seen that IPET does not allow straightforward, constructive top-down application.

4 On another binding function $Bind_{inh_{B0}}$ which complies with 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 leads to Theorem 19 analogous to Theorem 2 on satisfaction of BIPET's rules, by a function $Bind_{inh_{B0}}$ analogous to function $bind_{inh_0}$. As previously, we assume that every user declared class has an explicit extends clause with a type of length ≥ 1 as the authors of [IP02] and their calculus are requiring. $Bind_{inh}$ is to become a mapping

$$Bind_{inh} : Types\ \times\ \mathcal{C}^{RO}\ \longrightarrow\ (\mathcal{C}^{O} \cup \{null\})$$

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 $\mathcal{C}^{RO}\ is\ \mathcal{C}^{O} \cup \{Root\}$. $Types = \mathcal{CT}$ is the set of simple or qualified types over the set $\mathcal{SCT}$ of class identifiers occurring in program π. The function $Bind_{inh}$ is parameterized 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) order in the set $\mathcal{A}^*$ of all words or paths over $\mathcal{A}$. For example, the words $inh \prec decl^\frown inh \prec decl \prec inh^\frown decl$ are in this order. Notice, this order is total, but not well-founded!

Let $w = id_1^\frown id_2^\frown \cdots^\frown id_n,\ w \in \mathcal{A}^*,\ n \geq 0$. 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. Clear, the empty word λ yields $\lambda(P) = P$.

For a class $P \in \mathcal{C}^{RO}$ we have the set of admissible paths: A path w is called admissible in P if either w is empty or in case of non-

emptiness $w(P)$ is defined $\in \mathcal{C}^{RO}$ and all intermediate results, P included, $w(P)$ excepted, must be $\in dom_{inh} \cup \{Object\} = dom^{O}_{inh}$. So maximal admissible paths end up in $\mathcal{C}^{RO} \setminus dom_{inh}$.
Now, we have the following

Definition 3 *Let X be a type of length ≥ 1 , $P \in \mathcal{C}^{RO}$ and $inh \in \mathcal{C} \xrightarrow{part} \mathcal{C}^{O}$. Then*

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

Remark 4 *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 $inh^{i} \frown decl^{j}$, where $i, j \geq 0$ and in the induction step we restrict to words of the form inh^{i} where $i \geq 0$; this latter restriction guarantees admissibility implicitly. So, if we replace the first occurrence of $\mathcal{A}^*$ in the induction beginning by $inh^* \frown decl^*$ and the second occurrence of $\mathcal{A}^*$ in the induction step by inh^* then we have exactly the definition of $bind_{inh}(X \text{ in } P)$. The following deliberations until Theorem 19 inclusive hold also for $bind_{inh}$ instead of $Bind_{inh}$, mutatis mutandis.*
Further properly chosen subsets of $\mathcal{A}^\star$ lead to binding functions which are also uniformly defined for all programs. □

Now we are going to look for a specific inheritance function inh_{B0} such that $Bind_{inh_{B0}}$ satisfies all rules of BIPET. We go an analogous way as in [LSW09] and look for an appropriate functional $BDfl'$ such that inh_{B0} is the least fixed point. The natural functional

$$BDfl(inh)(P) \stackrel{df}{=} Bind_{inh}(ext(P) \; in \; decl(P))$$

is, unfortunately, not monotone and continuous in

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

where $\mathcal{C} \stackrel{part}{\longrightarrow} \mathcal{C}^O$ 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 *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

$$\emptyset = inh_\perp \subset BDfl(inh_\perp) \subset BDfl^2(inh_\perp) \not\subseteq BDfl^3(inh_\perp)$$

because

$BDfl(inh_\perp)(B) = A$, $BDfl(inh_\perp)(D) = \perp$, $BDfl(inh_\perp)(F) = B\$E$

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

$BDfl^3(inh_\perp)(B) = A$, $BDfl^3(inh_\perp)(D) = A\$C$, $BDfl^3(inh_\perp)(F) = A\$E \neq B\$E$ □

Example 5 convinces us to replace the functional $BDfl$ by another functional $BDfl'$. But first we introduce the notion of

State.

Definition 6 *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 6 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 the 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} is satisfied, restricted to dom_{inh} as a subset of $\mathcal{C}$. *inh* and/or its dependency relation Dep_{inh} may have cycles:

Definition 7 *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. [LSW09]):

Definition 8 *A Java-program is called* Well-Formed *iff there exists an inheritance function* inh_{WF} *which satisfies the following two conditions*
I_{B1}*) the function* 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

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

of inheritance functions which are States. The word "State" is chosen because our algorithm LSWA_B (defined quite analogously to LSWA in [LSW04,LSW09], see also Appendix) which determines the least fixed point inh_{B0} is 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 9 *$BDfl'$ is a monotonous functional (and consequently is continuous because $\mathcal{C} \xrightarrow{State} \mathcal{C}^O$ is finite).*

We need four Lemmas 10 to 14 for a proof of Theorem 9.

Lemma 10 *For every State inh, for every class $K \in dom^{RO}_{inh}$ and for every type X: If $decl(Bind_{inh}(X \text{ in } K)) \in dom^{RO}_{inh}$ then $Bind_{inh}(X|^i \text{ in } K) \in dom^{RO}_{inh}$ for $1 \leq i < length(X)$.*

Proof. Assume the thesis of the Lemma is wrong. Then there is a smallest i_0 with $1 \leq i_0 < length(X)$ and $Bind_{inh}(X|^{i_0} \text{ in } K) \notin dom^{RO}_{inh}$. 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 $Bind_{inh}(X \text{ in } K)$ is, due to definition of admissibility, 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! $\square$

Lemma 11 *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 $1 \leq i < length(X)$ $\;Bind_{inh_0}(X|^i \; in \; K) \in dom^{RO}_{inh_0}$
then for all $1 \leq i < length(X)$
$\quad Bind_{inh_0}(X|^i \; in \; K) = Bind_{inh}(X|^i \; in \; K)$
and either
$\quad Bind_{inh_0}(X \; in \; K) = Bind_{inh}(X \; in \; K) = M \in \mathcal{C}^O$
with $decl(M) \in dom^{R}_{inh_0}$ *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 $w_0 = id_{0m} \frown \cdots \frown id_{01}$ 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 $w = id_m \frown \cdots \frown id_1$, $id_i \in \mathcal{A}$: 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 3 does not differentiate between inh_0 and inh which the definition is based on.
(Induction step $length(X) > 1$)
Let for every $1 \leq i < length(X)$
$$Bind_{inh_0}(X|^i \; in \; K) \in dom^{RO}_{inh_0} \quad (\star).$$
Due to induction hypothesis and assumption $(\star)$ we have for all $1 \leq i < length(X) - 1$
$\quad Bind_{inh_0}(X|^i \; in \; K) = Bind_{inh}(X|^i \; in \; K) \in dom^{RO}_{inh_0}$

and

$Bind_{inh_0}(X|^{length(X)-1}\ in\ K) =$
$Bind_{inh}(X|^{length(X)-1}\ in\ K) = M' \in dom^O_{inh_0}.$

with $decl(M') \in dom^R_{inh_0}$. So we find a quite analogous situation as in the induction base and Definition 3 does not differentiate between inh_0 and inh. □

Lemma 12 *If inh is a State then the inheritance function* $inh' = BDfl'(inh)$ *is an extension of inh.*

Proof. Let $A \in dom_{inh}$. Then $A \neq Root$, $A \neq Object$, $decl(A) \in dom^R_{inh}, inh(A) \in dom^O_{inh}, inh(A) = Bind_{inh}(ext(A)\ in\ decl(A))$ 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 13 *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 14 *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 Lemma 10, 11 and 12.
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 10 and Lemma 11 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 15 *on direct and indirect successors of States:*
If in this proof of Lemma 14 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 $\mathcal{C} \xrightarrow{State} \mathcal{C}^O$. □

Proof. Of Theorem 9 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

$\alpha^B_{inh_1}(A) \wedge\ M = Bind_{inh_1}(ext(A)\ in\ decl(A))$

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 $M =$
$Bind_{inh_1}(ext(A)\ in\ decl(A)) \in dom^O_{inh_1}$. Lemma 10 and Lemma 11 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 $\alpha^B_{inh_1}(A)$: $A \neq Root$, $A \neq Object, decl(A) \in dom^R_{inh_1} \subseteq dom^R_{inh_2}$. So $\alpha^B_{inh_2}(A)$ is holding and $M = inh'_2(A)$.

□

The following remark shows modular confluence of the direct successorship relation $\prec^{DS}$.

Remark 16 *(on modular confluence)*
The relation $\prec^{DS}$ is modularly confluent, i.e. if $inh \prec^{DS} inh^A$ and $inh \prec^{DS} inh^B$ and $inh^A \neq inh^B$ then there is a common

direct successor State sst with $inh^A \prec^{DS} sst$ *and* $inh^B \prec^{DS} sst$.

$$\begin{array}{ccc} inh & \xrightarrow{\prec^{DS}} & inh^A \\ {\scriptstyle \prec^{DS}}\downarrow & & \downarrow{\scriptstyle \prec^{DS}} \\ inh^B & \xrightarrow[\prec^{DS}]{} & sst \end{array}$$

Notice, $sst = inh^{AB} = inh^{BA}$
(due to an easy calculation using Lemma 14).
If a $\prec^{DS}$*-chain*

$$inh_0 = sst_0 \prec^{DS} sst_1 \prec^{DS} sst_2 \prec^{DS} \cdots \prec^{DS} sst_n, n \geq 0,$$

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

$$BDfl'(inh) = inh \cup \bigcup_{inh \prec^{DS} sst} sst$$

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* $inh_\perp^{max}$ *is the least fixed point of* $BDfl'$*.*
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 Lemma 10 and 11 B)*

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

where $A \in \mathcal{C} \setminus dom_{inh}$ *with* $decl(A) \in dom_{inh}^R \subset \mathcal{C}^R$ *is the generating candidate for* inh^A*.*

□

From Theorem 9 and Remark 16 follows

Corollary 17 *The functional* $BDfl'$ *in*

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

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

$$inh_{B0} = \mu BDfl' = \bigcup_{i \in Nat_0} BDfl'^{\,i}(inh_\perp) = BDfl'^{\,\kappa}(inh_\perp)$$

which is

$$= \bigcup_{inh_\perp \stackrel{\prec}{=}^S inh} inh \;=\; inh_\perp^{max}.$$

If inh_{B0} *is total on* $\mathcal{C}$ *then* inh_{B0} *makes the program Well-Formed.*

Theorem 18 *If a program is Well-Formed then* inh_{WF} *and the least fixed point* inh_{B0} *are identical. Especially* inh_{WF} *is uniquely determined.*

Proof. inh_{WF} is a fixed point totally defined on $\mathcal{C}$ due to I_{B1} in Definition 8 and is enclosing the least fixed point inh_{B0}. If both were different then there were candidate classes $A \in \mathcal{C} \setminus dom_{inh_{B0}}$ of inh_{B0} and no one were generating a direct successor State of inh_{B0}. As $decl(A) \in dom^R_{inh_{B0}}$ there were a candidate class M of inh_{B0} such that $\langle A, M\rangle \in Dep_{inh_{WF}}$ due to Lemma 11 (Especially: In inh_{B0} is no permanent lack of a class to be inherited, compare [LSW09]). So $Dep_{inh_{WF}}$ had a cycle inside the finite set of candidates contrary to assumption I_{B2} in Definition 8. □

Theorem 19 *If the given Java-program is Well-Formed then the function* $Bind_{inh_{B0}}$ *complies with all six rules of BIPET.*

Proof. Due to construction of the least fixed point $inh_{B0} = inh_\perp^{max} = inh_{WF}$ by repeated successor States (algorithm LSWA$_B$) all paths w from $P \in \mathcal{C}^{RO}$ to $w(P) \in \mathcal{C}^{RO}$ are admissible ones in P and there are no repeated classes occurring. So $\mu w(P)$ is equal $\mu_\alpha w(P)$, and $\mu_\alpha w(P')$ in Definition 3 can be replaced by $Bind_{inh}(C\ in\ P')$ (inh is inh_{B0} in our present situation).
Verification of the rules I. and II. is the same as earlier in the proof of Theorem 2.
III. (BET-SimpEncl)
Consider a user declared class $P.C$. From the first premise

$Bind_{inh_{B0}}(D\ in\ P) = T$ we have that there exists a word μw_0 – the least word such that $\mu w_0(P).D = T$. From the definition of $Bind_{inh_{B0}}$

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

Since inh_{B0} satisfies property I_{B1} we can simplify the right-hand side of the equation to $Bind_{inh_{B0}}(D\ in\ inh_{B0}(P.C))$.
The third premise reads: $Bind_{inh_{B0}}(ext(P.C).D\ in\ P) = null$. From this premise we conclude 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 follows that the value of $\lambda(P.C).D$ is undefined. Now consider $Bind_{inh_{B0}}(D\ in\ P.C)$. Notice that the equality $(\mu w_0^\frown decl)(P.C).D = T$ holds. Making use of the observations based on the second and third premises we conclude that the word $\mu w_0^\frown decl$ is the least word μw such that the expression $\mu w(P.C).D$ is defined. Hence, $Bind_{inh_{B0}}(D\ in\ P.C) = T$.
IV. (BET-SimpSup)
$inh_{B0}(P.C) = Bind_{inh_{B0}}(ext(P.C)\ in\ P)$ 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 $T = Bind_{inh_{B0}}(ext(P.C).D\ in\ P) = Bind_{inh_{B0}}(D\ in\ inh_{B0}(P))$ and T is $\mu w(inh_{B0}(P.C)).D$. Because $P.C.D$ is undefined $\mu w^\frown inh_{B0}$ is the least path, denoted $\mu\tilde{w}$, such that $\mu\tilde{w}(P.C).D = T = Bind_{inh_{B0}}(D\ in\ P.C)$.
V. (BET-Long)
The conclusion is holding due to definition of $Bind_{inh_{B0}}$.
VI. (BET.LongSup)
$inh_{B0}(P'.D) = Bind_{inh_{B0}}(ext(P'.D)\ in\ P')$ is valid if $P'.D$ is a user declared class because condition I_{B1} is holding. Due to definition of $Bind_{inh_{B0}}$ we have $U = Bind_{inh_{B0}}(ext(P'.D).C\ in\ P') = Bind_{inh_{B0}}(C\ in\ inh_{B0}(P'.D))$ and U is $\mu w(inh_{B0}(P'.D)).C$. Because $P'.D.C$ is undefined $\mu w^\frown inh_{B0}$ is the least path, denoted $\mu\tilde{w}$, such that $\mu\tilde{w}(P'.D).C = U = Bind_{inh_{B0}}(C\ in\ P'.D) = Bind_{inh_{B0}}(X.C\ in\ P)$. □

Remark 20 *: An addition like the one to Theorem 2 cannot be proved due to the extra requirement of non-repeated classes on*

paths. In Theorem 2 the extra requirement is fulfilled implicitly.

Theorems 2 and 19 motivate to generalize the notion of well-formedness of a Java-program.

Definition 21 *If a binding function $Bindfn$ is such that its associated inheritance function*

$$inh(A) \stackrel{df}{=} Bindfn(ext(A) \text{ in } decl(A)) \text{ for } A \in \mathcal{C}$$

satisfies condition I_{B1} and I_{B2} of Definition 8 (replace $Bind_{inh_{WF}}$ by $Bindfn$) then the program is called well-formed w.r.t. $Bindfn$.

We may observe that the model $Bind_{inh_{B0}}$ can be calculated by an algorithm LSWA$_B$ (compare our motivation of the concept "State" behind Definition 8, Remark 16, Proof of Theorem 19 and Appendix). Hence we have two different algorithms to construct models of the IPET-calculus. Now we see that the statement "a straigthforward algorithm ...[IP02, 5.2.1, p.82]" is not justified at all. First of all the authors of [IP02] did not give any description of the algorithm. Second, there exist at least two algorithms with different results. The cited statement does not answer the question which one of the algorithms is the proper one.

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

The dilemma with IPET does not end with the statement 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 binding function can be defined 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, 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, which is well-formed w.r.t. $bind_{inh_0}$ and w.r.t. $Bind_{inh_{B0}}$, but the intersection function $extension(bind_{inh_0} \cap Bind_{inh_{B0}})$ does not comply with the rule III. (ET-SimpEncl), especially is different from $bind_{inh_0}$ which Java Language Specification JLS [GJSB05] has prescribed to be the appropriate binding function.

Lemma 22 *Program P of Example 5 has the following properties*

(i) Program P is well-formed w.r.t. $bind^P_{inh_0}$ *and program P is Well-Formed w.r.t.* $Bind^P_{inh_{B0}}$.

(ii) Both binding functions $bind^P_{inh_0}$ *and* $Bind^P_{inh_{B0}}$ *comply with all six rules of the system BIPET.*

(iii) The binding functions are not equal

$$bind^P_{inh_0} \neq Bind^P_{inh_{B0}}.$$

(iv) The intersection Int of two binding functions does not comply with the rules of BIPET system.

Proof. The proof of the lemma consists in exhibiting an example program of the previous section.
Example 5 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:

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

Now $bind_{inh_0}(\text{E } in\ B\$D) = B\$E$
because path *decl* from $B\$D$ to B is lexicographically less (from the right) than path $inh^\frown decl$ from $B\$D$ to A.

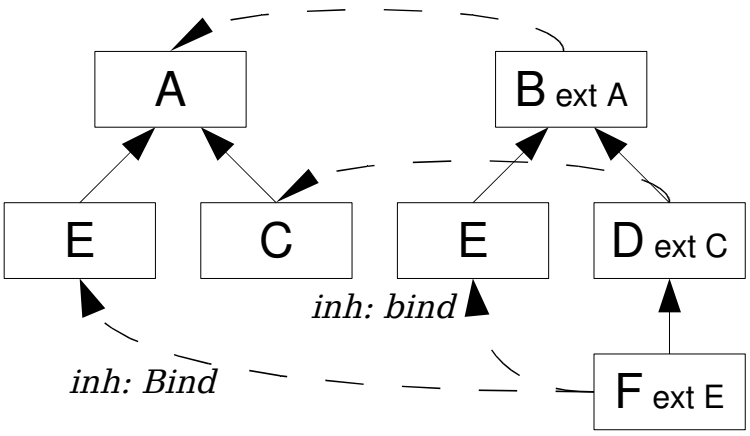

Fig. 1. according to function inh_0 the class $B\$D\$F$ inherits from the class $B\$E$, according to function inh_{B0} the class $B\$D\$F$ inherits from the class $A\$E$

But $Bind_{inh_{B0}}(\mathrm{E}\ in\ B\$D) = A\$E \neq B\$E$!
because path $decl^\frown inh$ from $B\$D$ to A is lexicographically less (from the right) than path $decl$ from $B\$D$ to B.
Let us define the intersection relation (function)

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

extension is the function which extends a partial function by *null*. Note that $Int(\mathrm{E}\ in\ B\$D)$ is *null* (*).
We shall prove that the Int-relation does not satisfy rule III.(ET-SimpEncl). From the facts gathered till now we deduce (using rule III.) that the value of $Int(\mathrm{E}\ in\ B\$D)$ should be $B\$E$:

$B = decl(B\$D)$

$C = ext(B\$D)$
$B\$D$.E is undefined
$Int(\text{E } in\ B) = B\$E$
$Int(\text{C.E } in\ B) = null$
because
$bind_{inh_0}(\text{C.E } in\ B) = null$
as $bind_{inh_0}(\text{C } in\ B) = A\$C$ has no attribute class named E
and
$Bind_{inh_{B0}}(\text{C.E } in\ B)$
$= Bind_{inh_{B0}}(\text{E } in\ Bind_{inh_{B0}}(\text{C } in\ B))$
$= Bind_{inh_{B0}}(\text{E } in\ A\$C)$
$= A\$E$.
So due to the assumed rule III. for Int
$Int(\text{E } in\ B\$D) = B\$E$
what is contradicting (*).
Hence Int is not a model of IPET.
□

So we have unexpectedly seen two different models of the BIPET-calculus such that their intersection is not a model of BIPET. Therefore BIPET cannot be treated as a direct or implicit definition of THE binding function.

Remark 23 *One might think that program presented in Example 22 of [LSW09] demonstrates this phenomenon already so that development of $Bind_{inh_{B0}}$ with its high expenditures might be superfluous. But that is not true. Sure, Example 22 has two different inheritance functions inh_1 and inh_2 which satisfy condition I_1. However condition I_2 is not satisfied. So $bind_{inh_1}$ and $bind_{inh_2}$ are two different models of BIPET due to the Addition to Theorem 2. But the program is not well-formed and the intersection property holds: This statement is true because extension($bind_{inh_1} \cap bind_{inh_2}$) is $bind_{inh_0}$ and $inh_0 = inh_1 \cap inh_2$ is the least fixed point of binding functional Bdfl′ resp. is algorithm LSWA's result. $bind_{inh_0}$ is a model of BIPET because inh_0 satisfies condition I_1' of the Addition to Theorem 2. So $bind_{inh_1}$ and $bind_{inh_2}$ of Example 22 do not satisfy those intriguing properties which*

the binding functions pair $bind_{inh_0}$ and $Bind_{inh_{B0}}$ of Example 5 satisfy.

Remark 24 *Our second remark stresses that one can imagine two compilers $C1$ and $C2$. One compiler is using the algorithm that computes the function $bind_{inh_0}$ and the second compiler uses the algorithm that computes the function $Bind_{inh_{B0}}$. Now, one can provide a program (an easy modification of the Example 5) such that two compilers give different results. Which result is a correct one? The IPET calculus will not provide an answer.*

We conclude that the immediate instinct to enrich BIPET by a (metatheoretic!) clause: "*choose the least of all BIPET's complying binding functions*" might not lead to any positive solution.

But we might conjecture that BIPET has at most one minimal complying solution instead of at most one least complying solution. A closer investigation of minimal complying binding functions shows: Confined to program Example 5 we can demonstrate that $bind_{inh_0}$ is minimal. $Bind_{inh_{B0}}$ is not minimal, but there is a minimal complying binding function $BINDFN'$ contained in $Bind_{inh_{B0}}$ which is different from the minimal $bind_{inh_0}$. So the conjecture above is wrong.

On the other hand: If there is exactly one minimal complying binding function then this is the least one. A lemma of this kind does not hold for a general partial order instead of the set of complying binding functions. A more thorough investigation finds out that $bind_{inh_0}$ is minimal for all well-formed programs. It is open whether there are other uniformly defined complying binding functions with this property.
Till now we have learned two families of binding functions. Both are defined uniformly with respect to Java programs. Below we show that there are infinitely many functions bind – each tailored with respect to a program such that they comply with respect to the rules of BIPET and differ from earlier defined binding functions $bind_{inh_0}$ and $Bind_{inh_{B0}}$.

- Remark, that there exists a program π and corresponding function $bind^{\pi}$ such that the function complies with the rules of BIPET and differs from *bind* and *Bind*.

  Example 25 *Program π consists of three top level classes.*
  class A extends Object {}
  class B extends Object {}
  class C extends Object {}

  The function $bind^{\pi}$ is defined on the base of the function $bind_{inh_0}$

  $$bind^{\pi}(X \ in \ P) = \begin{cases} Root.C & if \ X = A.B, \ P = Root \\ bind_{inh_0}(X \ in \ P) & otherwise \end{cases}$$

  We leave as an exercise the task of verification that the function $bind^{\pi}$ complies with the rules of BIPET.
  Hint. Verify whether the equality $bind(A.B \ in \ Root) = Root.C$ may appear as a conclusion or as a premise in any rule of BIPET. □
- Observe, that the value of $bind^{\pi}(A.B \ in \ Root) = Root.C$ and the name of resulting class is C and paradoxically is not equal B as one would expect.
- It is more or less obvious that there are infinitely many paradoxical examples with their wild functions *bind*.

6 Problem – how to define IPET in a correct way

In earlier sections we have seen that the calculus IPET can not be treated as a definition of function *bind* for it has many mutually contradicting "models". In this section we define criteria that a correct version of the system IPET should satisfy.
Let Π be the set of syntactically correct Java codes $\pi \in \Pi$, i.e. the codes that passed the syntacical analysis. Some of them are

correct static semantically or well-formed Java programs, some of them are not.

The task is to construct a uniform family $\{\mathcal{T}_\pi\}_{\pi\in\Pi}$ of theories. Each theory $\{\mathcal{T}_\pi\} = \langle \mathcal{L}, Cons, Axiom_\pi\rangle$ has the same language $\mathcal{L}$ and the same operation $Cons$ of logical consequence.

The theories are to define for each program π the binding function $bind^\pi$. We did it earlier in [LSW08], [LSW09] in terms of algebraic structure $\mathcal{S}$ of classes of program π. Now we need to reformulate the description of structure of classes.

Structures of Classes
Each program $\pi \in \Pi$ determines an algebraic structure $\mathcal{S}^\pi$ of classes of the program. Program π is finite. It determines the following sets:

- Id - the set of identifiers found in program π,
- $Classes$ - the set of class declarations in program π,
- $ClassTypes$ - the set of class types occurring in program π

and a function $extends$

$$extends:\ Classes \rightarrow\ ClassTypes$$

the diagram of this function is distributed through the declarations of classes.
For many reasons one can identify an occurrence of class declaration with the *absolute path* leading to the class declaration occurence, i.e. a finite sequence $c_1, \ldots, c_k,\ k > 1$ of identifiers such that

(1) c_1 is the identifier of a top-level class of the program,
(2) for each $1 < i \leq k$ the identifier c_i denotes a class declared as an inner class of the class $c_1, \ldots, c_{i-1}$
(3) c_k is the identifier of the class being declared.

Hence the set $Classes$ can be conceived as a finite set of sequences of identifiers closed with respect to prefixes, the empty sequence

is to be identified with the fictious class *Root* enclosing all top level classes.
It means that the set of *classes* forms a tree.

The set $ClassTypes$ is the set of finite sequences of identifiers separated by dots. Note, the phrase extends in a Java program allows to write a sequence of any length, repetitions of identifiers are allowed. Class types may appear after the keyword **extends** or in declarations of local fields of classes. Looking at three rules of calculus IPET we see that apart of class types appearing in the program the task of elaborating may need some class types that do not occur in the program. For all these reasons the set $\mathcal{CT}$ of class types is an infinite set of all finite sequences of identifiers separated by dots.

Remark 26 *For the readers who probably noted the difference between this definition of the structure of classes and the definition of structure of classes in [LSW09] we explain:*

(1) One can define the missing function decl as follows

$$decl(c_1, \ldots, c_k) \stackrel{df}{=} c_1, \ldots, c_{k-1},\ k > 1,\ decl(c_1) \stackrel{df}{=} Root.$$

(2) the partial function P.D is defined as follows: Let P be a class, D an identifier, the value of P.D is defined iff P.D is a class in the program.

(3) In the rules of IPET one does not need the function decl.

The goal is to give an axiomatic description of an extension of each structure $\mathcal{S}^\pi$ of classes by the function $bind^\pi$

$$bind^\pi :\ ClassTypes \times Classes^{RO} \to (Classes^O \cup \{null\})$$

in such a way that program fulfills criteria of being well-formed (in terms of [IP02], one says the program fulfills the sanity conditions) or to signal that no such function exists.

THEORIES
The language $\mathcal{L} = \langle \mathcal{A}, \mathcal{F} \rangle$ of each theory has the same alphabet $\mathcal{A}$ and the same set $\mathcal{F}$ of formulas. The alphabet contains variables of following sorts:

- $V_{\mathcal{CT}}$ – the set of variables of class types,
- $V_{\mathcal{C}}$ – the set of variables of type classes,

- $V_{\mathcal{I}}$ – the set of variables of class identifiers

and constants

- *null* - the exceptional pseudoobject value signalling error in program's structure,
- Id - the set of all identifiers

moreover the alphabet contains predicates *bind*, is_class and one functor ext and
auxiliary signs such as parentheses( ), colon, and the separator in.

The set $\mathcal{F}$ of formulas of theory $\mathcal{T}_\pi$ admits expressions of the form

- $bind(X\ in\ P) = T$ - the meaning of class type variable X inside class P is class T or *null*,
- $is\_class(X)$ - the value of class type argument X is a class,
- $ext(P) = X$ - the class type X is declared as an extension of class P.

The theory $\mathcal{T}_\pi$ is the union of the proto-theory $\mathcal{T}_0$ and the diagram of the structure of classes of program π.

Diagram Δ_π of the structure of classes of program π consists of

- finite set of formulas $is\_class(X)$, where X is a class type,
- a finite set of formulas $ext(P) = Y$, where P is a class and Y is a class type.

which constitute the full description of the structure of classes of program π.

The proto-theory $\mathcal{T}_0$ may resemble the set of inference rules BIPET or IPET. It will be useful in the definition of consequence operation (proof).

The axioms $Axiom_\pi$ of theory $\mathcal{T}_\pi$ are the formulas of diagram of the structure of classes of program π.

Let α be a formula $bind(X\ in\ P) = T$ where X is a class type of program π, $X \in ClassTypes$, $P, T \in Classes$. A *proof* of the formula α is a sequence of formulas $\beta_1, \beta_2, \ldots, \beta_m$, $m > 0$ such that

- formula β_m is α,
- for each $1 \leq i \leq m$ the formula β_i is
  - either an axiom or
  - it is the conclusion in one of inference rules given by prototheory $\mathcal{T}_0$ and all premises of the inference rule occur (earlier) in the sequence $\beta_1, \beta_2, \ldots, \beta_{i-1}$.

The *theorems* of theory $\mathcal{T}_\pi$ are exactly these formulas of the form $bind(X\ in\ P) = T$ which possess a proof in theory $\mathcal{T}_\pi$.

A *model* $\mathfrak{M}_\pi$ of theory $\mathcal{T}_\pi$ is a pair $\langle \Delta_\pi, bind^\pi \rangle$ consisting of the diagram Δ_π of program π and a function $bind^\pi$ such that all theorems of theory $\mathcal{T}_\pi$ are valid in $\mathfrak{M}_\pi$. Since the validity of axioms is assured by the definition it remains to be checked that for each inference rule if the premises are valid then the conclusion of the rule is valid too.

A *uniform model* $\mathfrak{M}_\Pi$ of the family $\{\mathcal{T}_\pi\}_{\pi \in \Pi}$ of theories with respect to the diagrams of programs $\pi \in \Pi$ is a function $bind_\Pi$, shortly $bind$,

$$bind : \Pi \times ClassTypes \times Classes^{RO} \to \{Classes^O \cup \{null\}\}$$

such that for every $\pi \in \Pi$ function $bind^\pi$, defined on the base of function $bind$ by the following equality

$$bind^\pi(X, P) = bind(\pi, X, P) \text{ for every } X \in \mathcal{CT} \text{ and } P \in \mathcal{C}^{RO}\ ,$$

together with the diagram Δ_π of program π makes a model of theory $\mathcal{T}_\pi$.

Each theory $\mathcal{T}_\pi$ should enjoy a few metamathematical properties:

(1) (*categoricity*) the theory possesses one model (up to isomorphisms), moreover the function $bind_{inh_0}$ is the model,

(2) (*decidability*) the theory is decidable,
(3) (*low complexity*) the complexity of the algorithm deciding on the truth of a formula should not be higher than the complexity of the $bind_{inh_0}$ algorithm.

Every theory $\{\mathcal{T}_\pi\}_{\pi\in\Pi}$ must be decidable. This requirement is very strong, however indispensable. It seems reasonable to add the requirement that the algorithm deciding of the validity of formulas $bind(X\ in\ P) = T$ is of complexity comparable to the complexity of the algorithm for calculating function $bind_{inh_0}$.

7 Concluding remarks

The identification of a declarative occurrence T of a class which is binding an applied occurrence of a (class) type X within a class P is basic for the understanding how a program works. The paper [IP02] offers the IPET-calculus for deducing the values of the function $bind(X \text{ in } P) = T$, in the original paper it is written $P \vdash X \Rightarrow T$. It turned out that the formal system of IPET has many models, hence, the system does not define the binding function.

The discussion of the present paper shows how important it 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 in 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 calculus IPET 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, which binds an applicative occurrence of type X inside class P to the declaration T of a class.
(2) (*categoricity or completeness*) How many models has a pro-

posed formal system? In our case the question is: *are there different functions bindfn which are models of the IPET-calculus?* The positive answer tells us that something important has escaped our attention, in our case the existence of the different models $bind_{inh_0}$ and $Bind_{inh_{B0}}$.

(3) (*repairing an incomplete system*) If there are several models, one should try to repair the formal specification either by adding and changing axioms and inference rules (this way, we believe, is the correct one; so we have presented calculus BIPET' in a forthcoming article) 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 a certain algorithm.

These questions were not addressed in paper [IP02].

A few words on the problem formulated in the previous section

We suspect that these metamathematical requirements (categoricity and decidability) imposed on the goal of constructing a simple theory of binding identifiers are contradicting themselves. For the requirement that the theory must be decidable contradicts the fact that every first-order decidable theory that has infinite model has also non-standard models c.f. [MSST2001].

We stop here with one additional remark: one should consider the requirement that the formal theory of binding should allow to distinguish between well-formed programs and these which are not well-formed. The present authors do not know how to formulate an appropriate condition in terms of metamathematics. A candidate formulation like: "*if there exists a type X and class P such that the formula $\ulcorner bind_\pi(X, P) = null\urcorner$ has a proof then the program π is not well-formed (does not satisfy the sanity conditions)*" is far from being satisfactory. History of implementations of programming languages since 1960 has shown that decent understanding of the meanings of nested program struc-

tures is a great problem, not only for users, but even for language designers and compiler builders who are expected to have a higher education in informatics than users. A thorough pervasion of static binding of names, most natural since the origins of predicate logic and lambda calculus, by concepts of theoretical informatics, mathematics and mathematical logics is an absolute must. The more theoretical knowledges of binding we have the higher is the chance that both – users and compilers – conceive program semantics in the same manner. Strong theoretical connections assure that ideas of programming language designers and practicioners will achieve lasting importance.

Acknowledgement. We would like to thank the anonymous reviewers of article [LSW09] who have encouraged us to write a full paper on our observations of types elaboration in Java with inner classes in Igarashi's and Pierce's article [IP02].

Appendix: An algorithm for binding function $Bind_{inh}$

In order to assure that LSWA$_B$ is really an algorithm it is sufficient to present a subalgorithm which for all arguments $X \in Types$ and $P \in dom^{RO}_{inh}$ either terminates successfully (regularly) with result $Bind_{inh}(X\ in\ P) \in dom^{O}_{inh}$ or otherwise terminates with an error report. That is correct because for every candidate class K its father class $decl(K)$ is from dom^{R}_{inh}. So we name the restricted binding function to be computed $Bind^{restr}_{inh}$:

$Bind^{restr}_{inh}(X\ in\ P) =$
var $\mathcal{C}^{ROU} P', int\ i;$
$i := 0;\ P' := P;$
while $i < length(X)$
do $i := i + 1;$
$\quad P' := Bind^{simple}_{inh}(X_i\ in\ P');$
$\quad$ **if** $P' \notin dom^{O}_{inh}$

then *error*
fi
endwhile ;
result :=P'
end $Bind^{restr}_{inh}$

$Bind^{simple}_{inh}(C\ in\ P) =$
var $\mathcal{C}^{ROn}\ T$; // $\mathcal{C}^{ROn} = \mathcal{C}^{RO} \cup \{null\}$
if $P.C$ is defined $\in \mathcal{C}^{O}$, i.e. $\neq null$
then $P.C$
else if $P \in dom_{inh}$
then $T := Bind^{simple}_{inh}(C\ in\ inh(P))$;
if T is defined $\in \mathcal{C}^{O}$, i.e. $\neq null$
then T
else $Bind^{simple}_{inh}(C\ in\ decl(P))$
fi
else
if $P = Object$
then $Bind^{simple}_{inh}(C\ in\ Root)$
else $null$
fi
fi
fi
end $Bind^{simple}_{inh}$

References

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

[Bjo09] D.Bjoerner. Domain Engineering – Technology Management, Research and Engineering. COE Research Monograph Series, Vol. 4, JAIST Japan, 2009

[DaNy67] 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

[Dij60] E.W. Dijkstra. Recursive Programming. Numerische Mathematik **2**, 312-318, 1960

[GHL67] A.Grau, U.Hill, H.Langmaack. Translation of ALGOL60. Handbook for Automatic Computation, Vol. I, Part b (chief ed. K.Samelson), Springer 1967

[GJS96] J. Gosling, B. Joy, G. Steele. The Java Language Specification. First edition, Addison-Wesley 1996

[GJSB00] J. Gosling, B. Joy, G. Steele. The Java Language Specification. Second edition, Addison-Wesley 2000

[GJSB05] J. Gosling, B. Joy, G. Steele, G. Bracha. The Java Language Specification. Third edition, Addison-Wesley 2005

[GoRo89] A.Goldberg, D.Robson. Smalltalk-80 – The Language. Addison-Wesley 1989

[Her65] H. Hermes. Enumerability, Decidability, Computability. Academic Press & Springer, NewYork & Berlin, Heidelberg, 1965

[Ich80] J.D. Ichbiah. Ada Reference Manual. LNCS 106, Springer-Verlag, Berlin, Heidelberg, New York 1980

[IP02] A. Igarashi, B. Pierce. On inner classes. Information and Computation **177**, 56-89, 2002

[JeWi75] K.Jensen, N.Wirth. Pascal, User Manual and Report, 2nd ed.. Springer 1975

[Kan74] P.Kandzia. On the "most recent"-property of ALGOL-like programs. In Proc. 2nd Coll. Automata Languages and Programming (J.Loeckx, ed.). LNCS 14, 97-111. Berlin, Heidelberg, New York, Springer 1974

[KSW88] A.Kreczmar, A.Salwicki, M.Warpechowski. Loglan'88 - Report on the Programming Language. LNCS 414, Springer, Berlin 1990

[Lan73] H.Langmaack. On Correct Procedure Parameter Transmission in Higher Programming Languages. Acta Informatica **2**, 2, 110-142, 1973

[LoSi84] J. Loeckx, K. Sieber. The Foundation of Program Verification. Wiley-Teubner 1984

[LSW04] H.Langmaack, A. Salwicki, M.Warpechowski. On correctness and completeness of an algorithm determining inherited classes and on uniqueness of solutions. In: G.Lindemann et al., Proc. CS&P'2004, Caputh Sept. 24-26, Vol. 2, 319-329, Informatik-Berichte, Humboldt Univ. Berlin, 2004

[LSW08] H.Langmaack, A.Salwicki, M.Warpechowski. On an deterministic algorithm identifying direct superclasses in Java, Fundamenta Informaticae **85**, 343-357, 2008

[LSW09] 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**, 389-410, 2009

[McC+65] J.McCarthy et al.. LISP 1.5 Programmer's Manual. The M.I.T.Press, Cambridge, Mass., 1965

[McG72] C.L.McGowan. The "most recent"-error: its causes and correction. In: Proc. ACM Conf. on Proving assertions about programs. SIGPLAN Notices 7, No.1, 191-202, 1972

[MMPN93] O.L.Madsen, B.Moeller-Pedersen, K.Nygaard. Object Oriented Programming in the BETA Programming Language. Addison Wesley / ACM Presss, 1993, see also: Beta Programming Language, 2001, available from: http://www.daimi.au.dk/∼beta/

[MSST2001] G.Mirkowska, A.Salwicki, M. Srebrny, A. Tarlecki. First-order Specifications of Programmable Data Types, SIAM Journal on Computing, **30**, pp. 2084-2096, 2001

[Nau63] P.Naur (ed.) et al.. Revised Report on the Algorithmic Language ALGOL60. Num. Math. **4**, 420-453, 1963

[Old81] E.R.Olderog. Charakterisierung Hoarescher Systeme für ALGOL-ähnliche Programmiersprachen. Dissertation, Inst. F. Informatik u. Prakt. Math., Univ. Kiel, Bericht 5/81, 1981

[Plo77] G.D.Plotkin. LCF Considered as a Programming Language. Theoretical Computer Science **5**, 223-255, 1977

[RS63] H. Rasiowa, R. Sikorski, Mathematics of metamathematics, PWN Publ., Warsaw, 1963

[Ste84] G.L.Steele jr.. Common LISP - The Language. Digital Press 1984

[Wij+68] A.van Wijngaarden et al. (eds.). Report on the Algorithmic Language ALGOL68. Numerische Mathematik **14**, 79-218, 1969