Java is Type Safe — Probably

Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
email: sd and se @doc.ic.ac.uk

Abstract. Amidst rocketing numbers of enthusiastic Java programmers and internet applet users, there is growing concern about the security of executing Java code produced by external, unknown sources. Rather than waiting to find out empirically what damage Java programs do, we aim to examine first the language and then the environment looking for points of weakness. A proof of the soundness of the Java type system is a first, necessary step towards demonstrating which Java programs won't compromise computer security.
We consider a type safe subset of Java describing primitive types, classes, inheritance, instance variables and methods, interfaces, shadowing, dynamic method binding, object creation, null and arrays. We argue that for this subset the type system is sound, by proving that program execution preserves the types, up to subclasses/subinterfaces.

1 Introduction

Before the first complete Java language description was available [13] use of the language was extremely widespread and the rate of increase in usage is steep. The language may not have reached a stable point in its development yet: there exist differences between the language descriptions [16, 17, 13], and there are many suggestions for additional features [19, 2]. Several studies have uncovered flaws in the security of the Java system [11], and have pointed out the need for a formal semantics.
Java combines the experience from the development of several object oriented languages, such as C++, Smalltalk and CLOS. The main features of the language are primitive types (character, integer, boolean, float), classes with inheritance, instance/class variables and methods, interfaces for class signatures, shadowing of instance variables, dynamic method binding, exceptions, arrays, strings, class modifiers (`private`, `protected`, `public` *etc*), `final`/`abstract` classes and methods, nested scopes, separate compilation, constructors and finalizers. The philosophy of the language designers was to include only features with already known semantics, and to provide a small and simple language.
Nevertheless, we feel that the introduction of some new features in Java, as well as the specific combination of features, justifies a study of the Java formal semantics. The use of interfaces, reminiscent of [9, 5] is a simplification of the signatures extension for C++ [3] and is – to the best of our knowledge – novel. The mechanism for dynamic method binding is that of C++, but we know of

no formal definition. Java adopts the Smalltalk approach whereby all object variables are implicitly pointers.
Furthermore, although there are a large number of studies of the semantics of isolated programming language features or of minimal programming languages [1], there have not been many studies of the formal semantics of *actual* programming languages . In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.
Experience confirms the importance of formal studies of type systems early on during language development. Eiffel, a language first introduced in 1985, was discovered to have a loophole in its type system in 1990 [8, 18]. Given the growing usage of Java, it seems important that if there are loopholes in the type system they be discovered early on.
We aim to argue that the type system of Java is sound, in the sense that unless an exception is raised, the evaluation of any expression will produce a value of a type "compatible" with the type assigned to it by the type system.
We were initially attracted to Java, because of its elegant combination of several tried language features. For this work we were guided by the language descriptions, [17], [13]. We found the language description complete and unambiguous, in the sense that any question relating to semantics could be answered unambiguously by [13]. However, we discovered some rules to be more restrictive than necessary, and the reasons for some design decisions were not obvious. We hope that the language authors will publish a language design rationale soon.

1.1 The Java subset considered so far

In this paper we consider the following parts of the Java language: primitive types, classes and inheritance, instance variables and instance methods, interfaces, shadowing of instance variables, dynamic method binding, object creation with `new`, the `null` value, arrays, and some exceptions[12].
We chose this Java subset because we consider the Java way of combining classes, interfaces and dynamic method binding to be both novel and interesting. Furthermore, we chose an imperative subset right from the start, because the extension of type systems to the imperative case has sometimes uncovered new problems, (*e.g.* multi-methods for functional languages [7], and for imperative languages in[4], the Damas and Milner polymorphic type systems for functional languages [10], and for the imperative extension [21]). We considered arrays, because of the known requirement for run time type checking.
We describe the language as in the [13] definition with the exception of method binding, which we model as described in [17], because it imposes a weaker requirement. Namely, [17] requires methods that hide methods from superclasses or superinterfaces to have a return type that can be widened to the return type of the hidden method, whereas [13] requires them to have the same type. Because the first requirement is weaker, our soundness result automatically applies to the new, stricter version of Java as in [13].

1.2 Our approach

We define Java$_s$, a safe subset of Java containing the features listed previously, a term rewrite system to describe the Java$_s$ operational semantics and a type inference system to describe compile-time type checking. We prove that program execution preserves the types up to the subclass/subinterface relationship.
We aimed to keep the description straightforward, and so we have removed some of the syntactic sugar in Java, *e.g.* we require instance variable access to have the form `this.var` as opposed to `var`, and we require the last statement in a method to be a `return` statement. These restrictions simplify the type inference and term rewriting systems.
The type system is described in terms of an inference system. In contrast with many type systems for object oriented languages, it does not have a subsumption rule, a crucial property when type checking message expressions, *c.f.* 3.2. Contrary to Java, Java$_s$ statements have a type – and thus we can type check the return values of method bodies.
The execution of Java programs requires some type information at run-time (*e.g.* method descriptors as in ch. 15.11 in [13]). For this reason, we define Java$_{se}$, an enriched version of Java$_s$ containing compile-time type information to be used for method call and field access. Interestingly, it turns out, that in contrast to Java and Java$_s$, Java$_{se}$ *does* enjoy a "substitution property". Hence in Java$_{se}$ the replacement of a subexpression of type `T` by another subexpression of a subtype of `T`, does not affect the type of the overall expression – up to the subclass/subinterface relationship. This should not be surprising, since the lack of a substitution property in Java was probably the reason for the introduction of method descriptors in the first place.
The operational semantics is defined for Java$_{se}$ as a ternary rewrite relationship between configurations, terms and configurations. Configurations are tuples of terms and states. The terms represent the part of the original program remaining to be executed. We describe method calls through textual substitution.
We have been able to avoid additional structures such as program counters and higher order functions. The Java$_s$ simplifications of eliminating block structure and local variables allow the definition of the state as a flat structure, where addresses are mapped to objects and global variables are mapped to primitive values or addresses. Objects carry their classes (similar to the Smalltalk abstract machine [15], thus we do not need store types [1], or location typings [14]). Objects are labelled tuples, where each label contains the class in which it was declared. Array values are tuples too, and they are annotated by their type and their dimension.
This paper is organized as follows: In section 2 we give the syntax of Java$_s$. In section 3 we define the static types for Java$_s$, and the mapping from Java$_s$ to Java$_{se}$. In section 4 we describe states, configurations and the operational semantics for Java$_{se}$. In section 5 we prove the Subject Reduction Theorem. In section 6 we give an example. Finally, in section 7 we outline further work and draw some conclusions.

2 The language Java$_s$

Java$_s$ describes a subset of Java, including classes, instance variables, instance methods, inheritance of instance methods and variables, shadowing of instance variables, interfaces, widening, method calls, assignments, object creation and access, the null value, instance variable access and the exception NullPointExc, arrays, array creation and the exceptions ArrStoreExc, NegSzeExc and IndOutBndExc. We have not yet considered initializers, constructors, finalizers, class variables and class methods, local variables, class modifiers, final/abstract classes and methods, `super`, strings, numeric promotions and widenings, concurrency, the handling of exceptions, packages and separate compilation.
There are slight differences between the syntax of Java$_s$ and Java which were introduced to simplify the formal description. A Java program contains both type and evaluation information. The type information consists of variable declarations, parameter and result types for methods, and interfaces of classes. The evaluation information consistent statements in method bodies. In Java$_s$ this information is split into two: type information is contained in the environment (usually represented by a Γ), whereas evaluation information is reflected in the program (usually represented by a `p`). An example can be seen in section 6.
We follow the convention that Java$_s$ keywords appear as keyword, identifiers as `identifier`, nonterminals appear in italics as *Nonterminal*, and the metalanguage symbols appear in roman (*e.g.* ::=, (,*,)). Identifiers with the suffix `Id` (*e.g.* `VarId`) indicate the identifiers of newly declared entities, whereas identifiers with the suffix `Name` (*e.g.* `VarName`) indicate a previously declared entity.

2.1 Programs

A program consists of a sequence of class bodies. Class bodies consist of a sequence of method bodies.
Method bodies consist of the method identifier, the names and types of the arguments, and a statement sequence. We require that there is exactly one return statement in each method body, and that it is the last statement. This simplifies the Java$_s$ operational semantics without restricting the expressiveness, since it requires at most a minor transformation to enable any Java method body to satisfy this property.
We need only consider conditional statements, assignments and method calls. This is because loop, break, continue and case statements can be coded in terms of conditionals and loops; try and throw statements belong to exceptions which are outside the scope of the current state of our investigations.
We consider values, method calls, and instance variable access. Java values are primitive (*e.g.* literals such as true, false, 3, 'c' *etc*), references or arrays. References are null, or pointers to objects. The expression new C creates a new object of class C, whereas the expression new $\mathtt{T}[\mathtt{m}_1]...[\mathtt{m}_n][]_1...[]_k$, $\mathtt{n}+\mathtt{k} \geq 1$ creates a n+k-dimensional array value. Pointers to objects are implicit. We distinguish variable types (sets of possible run-time values for variables) and method types, as can be seen in figure 1.

$$
\begin{array}{lll}
\mathit{Program} & ::= & \{\ (\ \mathit{ClassBody}\ )^*\ \} \\
\mathit{ClassBody} & ::= & \texttt{ClassId}\ \ \mathsf{ext}\ \texttt{ClassName}\ \{\ (\ \mathit{MethodBody}\ )^*\ \} \\
\mathit{MethodBody} & ::= & \texttt{MethId}\ \ \mathsf{is}\ (\lambda\ \texttt{ParId}\ :\ \mathit{VarType}.)^*\ \{\ \mathit{Stmts}\ ;\ \mathsf{return}\ [\mathit{Expr}]\ \ \} \\
\mathit{Stmts} & ::= & \epsilon \mid \mathit{Stmts}\ ;\ \mathit{Stmt} \\
\mathit{Stmt} & ::= & \mathsf{if}\ \mathit{Expr}\ \mathsf{then}\ \mathit{Stmts}\ \mathsf{else}\ \mathit{Stmts} \\
 & \mid & \mathit{Var} := \mathit{Expr} \\
 & \mid & \mathit{Expr} \\
\mathit{Expr} & ::= & \mathit{Value} \\
 & \mid & \mathit{Var} \\
 & \mid & \mathit{Expr}.\texttt{MethName}\ (\ \mathit{Expr}^*) \\
 & \mid & \mathsf{new}\ \texttt{ClassName} \\
 & \mid & \mathsf{new}\ \mathit{SimpleType}\ ((\texttt{[}\ \mathit{Expr}\ \texttt{]})^+(\texttt{[}\ \texttt{]})^*)\ \mid(\texttt{[}\ \texttt{]})^+) \\
\mathit{Var} & ::= & \texttt{Name} \\
 & \mid & \mathit{Var}.\texttt{VarName} \\
 & \mid & \mathit{Var}[\mathit{Expr}] \\
 & \mid & \mathsf{this} \\
\mathit{Value} & ::= & \mathit{PrimValue} \mid \mathsf{null} \\
\mathit{PrimValue} & ::= & \mathit{intValue} \mid \mathit{charValue} \mid \mathit{byteValue} \mid \ldots \\
\mathit{VarType} & ::= & \mathit{SimpleVarType} \mid \mathit{ArrayType} \\
\mathit{SimpleType} & ::= & \mathit{PrimType} \mid \texttt{ClassName} \mid \texttt{InterfaceName} \\
\mathit{ArrayType} & ::= & \mathit{SimpleType}\texttt{[}\ \texttt{]} \mid \mathit{ArrayType}\texttt{[}\ \texttt{]} \\
\mathit{PrimType} & ::= & \mathsf{bool} \mid \mathsf{char} \mid \mathsf{int} \mid \ldots \\
\mathit{MethType} & ::= & \mathit{ArgType} \rightarrow (\mathit{VarType} \mid \mathit{void}) \\
\mathit{ArgType} & ::= & (\mathit{VarType}\ (\times \mathit{VarType})^*)
\end{array}
$$

Fig. 1. Java$_s$ programs

2.2 The environment

The environment, usually denoted by a Γ, contains both the subclass and interface hierarchies and variable type declarations. It also contains the type definitions of all variables and methods of a class and its interface. *StandardEnv* should include all the predefined classes, *e.g.* `Object` and all the classes described in chapters 20-22 of [13], but at the moment it is empty. Declarations consist of class declarations, interface declarations and identifier declarations.

A class declaration introduces a new class as a subclass of another class (if no explicit superclass is given, then `Object` will be assumed), a sequence of component declarations, and optionally, interfaces implemented by the class. Component declarations consist of field identifiers and their types, and method identifiers and their signatures. Method bodies are not declarations; they are found in the program part rather than the environment.

An interface declaration introduces a new interface as a subinterface of several other interfaces and a sequence of components. The only interface components in Java$_s$ are methods, because interface variables are implicitly static, and we have

$$
\begin{array}{lll}
\mathit{Env} & ::= & \mathit{StandardEnv} \mid \mathit{Env}\ ;\ \mathit{Decl} \\
\mathit{StandardEnv} & ::= & \epsilon \\
\mathit{Decl} & ::= & \texttt{ClassId ext ClassName impl } (\texttt{InterfName})^* \\
 & & \quad \{\ (\texttt{VarId} : \mathit{VarType})^*\ (\texttt{MethId} : \mathit{MethType})^*\ \} \\
 & \mid & \texttt{InterfId ext InterfName}^* \{\ (\texttt{MethId} : \mathit{MethType})^*\ \} \\
 & \mid & \texttt{VarId} : \mathit{VarType}
\end{array}
$$

Fig. 2. Java$_s$ environments

not yet considered static variables. Variable declarations introduce variables of a given type.

$$
\frac{\Gamma = \Gamma',\ \texttt{C ext C}'\ \texttt{impl}\ \ldots\{\ldots\}, \Gamma''}{\Gamma \vdash \texttt{C} \sqsubseteq \texttt{C}}
\qquad
\frac{}{\vdash \texttt{Object} \sqsubseteq \texttt{Object}}
$$

$$
\frac{\Gamma \vdash \texttt{C} \sqsubseteq \texttt{C}' \qquad \Gamma = \Gamma',\ \texttt{C ext C}'\ \texttt{impl}\ \ldots \texttt{I} \ldots \{\ \ldots\ \}, \Gamma''}{\Gamma \vdash \texttt{C} :_{imp} \texttt{I}}
\qquad
\frac{\Gamma \vdash \texttt{C} \sqsubseteq \texttt{C}' \qquad \Gamma \vdash \texttt{C}' \sqsubseteq \texttt{C}''}{\Gamma \vdash \texttt{C} \sqsubseteq \texttt{C}''}
$$

$$
\frac{\Gamma = \Gamma'\ \texttt{I ext}\ \ldots,\ \texttt{I}', \ldots \{\ \ldots\ \}, \Gamma''}{\Gamma \vdash \texttt{I} \leq \texttt{I}'}
\qquad
\frac{\Gamma \vdash \texttt{I} \leq \texttt{I}' \qquad \Gamma \vdash \texttt{I}' \leq \texttt{I}''}{\Gamma \vdash \texttt{I} \leq \texttt{I}''}
$$

Fig. 3. subclass and subinterface relationships

The subclass $\sqsubseteq$ and the implements $:_{imp}$ relations are defined by the inference rules in figure 3. Every class introduced in Γ is its own subclass, and the assertion $\Gamma \vdash \texttt{C} \sqsubseteq \texttt{C}$ indicates that `C` is defined in the environment Γ as a class. The direct superclass of a class is indicated in its declaration. `Object` is a predefined class. The assertion $\Gamma \vdash \texttt{C} :_{imp} \texttt{I}$ indicates that the class `C` was declared in Γ as providing an implementation for interface `I`. The subclass relationship is transitive. Every interface is its own subinterface and the assertion $\Gamma \vdash \texttt{I} \leq \texttt{I}$ indicates that `I` is defined in the environment Γ as an interface. The superinterface of an interface is indicated in its declaration. The subinterface relationship is transitive.

Definition 1 *For a method type* $\texttt{MT} = \texttt{T}_1 \times \ldots \times \texttt{T}_n \rightarrow \texttt{T}$*, we define the* argument types *and the* result type*:*

– $Args(\mathtt{MT}) = \mathtt{T}_1 \times \ldots \times \mathtt{T}_n$
– $Res(\mathtt{MT}) = \mathtt{T}$

Variable types (*i.e.* primitive types, interfaces, classes and arrays) are required in type declarations; method types (*i.e.* n argument types, and a result type, with $\mathtt{n} \geq 0$) are required in method declarations. The assertion $\Gamma \vdash \mathtt{T} \diamond_{VarType}$ means that T is a variable type, $\Gamma \vdash \mathtt{AT} \diamond_{ArgType}$ means that AT is a method argument type, and $\Gamma \vdash \mathtt{MT} \diamond_{MethType}$ means that MT is a method type.

$$\frac{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}}{\Gamma \vdash \mathtt{C} \diamond_{VarType}} \qquad \frac{\Gamma \vdash \mathtt{I} \leq \mathtt{I}}{\Gamma \vdash \mathtt{I} \diamond_{VarType}} \qquad \frac{\Gamma \vdash \mathtt{T} \diamond_{VarType}}{\Gamma \vdash \mathtt{T}[] \diamond_{VarType}}$$

$$\frac{}{\begin{array}{l}\vdash \mathtt{int} \diamond_{VarType} \\ \vdash \mathtt{char} \diamond_{VarType} \\ \vdash \mathtt{bool} \diamond_{VarType}\end{array}} \qquad \frac{\begin{array}{l}\Gamma \vdash \mathtt{T} \diamond_{VarType} \\ \Gamma \vdash \mathtt{T_i} \diamond_{VarType} \quad \mathtt{i} \in \{1, \ldots \mathtt{n}\}, \mathtt{n} \geq 0\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \diamond_{ArgType} \\ \Gamma \vdash \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \rightarrow \mathtt{T} \diamond_{MethType}\end{array}}$$

Fig. 4. method and variable types

The widening relationship, described in figure 5, exists between variable types. If a type T can be widened to a type T′ (expressed as $\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{T}'$), then a value of type T can be assigned to a variable of type T′ without any run-time casting or checking taking place. This is defined in chapter 5.1.4 [17]; chapter 5.1.2 in [17] defines widening of primitive types, but here we shall only be concerned with widening of references. Furthermore, for the null value, we introduce the type nil which can be widened to any array, class or interface.

$$\frac{\Gamma \vdash \mathtt{T} \diamond_{VarType}}{\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{T}} \qquad \frac{\Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{T}'}{\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{T}'} \qquad \frac{\Gamma \vdash \mathtt{T} \leq \mathtt{T}'}{\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{T}'} \qquad \frac{\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{Object}}{\Gamma \vdash \mathtt{nil} <_{wdn} \mathtt{T}}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{T} \leq \mathtt{T} \\ or\ \Gamma(\mathtt{T}) = \mathtt{T}'[]\end{array}}{\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{Object}} \qquad \frac{\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{T}'}{\Gamma \vdash \mathtt{T}[] <_{wdn} \mathtt{T}'[]} \qquad \frac{\begin{array}{l}\Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{T}' \\ \Gamma \vdash \mathtt{T}' :_{imp} \mathtt{T}'' \\ \Gamma \vdash \mathtt{T}'' \leq \mathtt{T}'''\end{array}}{\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{T}'''} \qquad \frac{}{\Gamma \vdash \mathtt{nil} <_{wdn} \mathtt{nil}}$$

Fig. 5. widening relationship

2.3 Well-formed declarations and environments

It is easy to see that the relations $\sqsubseteq$, $:_{imp}$, $\leq$ and $<_{wdn}$ are computable for any environment. In this section we describe the Java requirements for variable, class and interface declarations to be well-formed. We indicate by $\Gamma \vdash \Gamma' \diamond$, that the declarations in environment Γ' are well-formed, under the declarations of the larger environment Γ. We need to consider a larger environment Γ because Java allows forward declarations (*e.g.* in section 6 the class `Phil` uses the class `FrPhil` whose declaration follows that of `Phil`). We shall call Γ well-formed, iff $\Gamma \vdash \Gamma \diamond$. Therefore, the assertion $\Gamma \vdash \Gamma' \diamond$ is checked in two stages: The first stage establishes the relations $\sqsubseteq$, $:_{imp}$, $\leq$ and $<_{wdn}$ for the complete environment Γ, and the second stage establishes that the declarations in Γ' are well-formed one by one, according to the rules in this section. Not surprisingly, the empty environment is well-formed. We need the notion of definition table lookup, *i.e.* $\Gamma(\mathtt{Id})$,

$$\frac{}{\Gamma \vdash \epsilon \diamond} \qquad \frac{\Gamma \vdash \Gamma' \diamond \quad \Gamma \vdash \mathtt{T} \diamond_{VarType} \quad \Gamma'(\mathtt{x}) = \mathsf{Undef}}{\Gamma \vdash \Gamma', \mathtt{x} : \mathtt{T} \diamond}$$

Fig. 6. well-formed declarations

which returns the definition of the identifier `Id` in Γ, if it has one.

Definition 2 *For an environment Γ, with unique definitions for every identifier, we define $\Gamma(\mathtt{id})$ as follows:*

- $\Gamma(\mathtt{x}) = \mathtt{T}$ *iff* $\Gamma = \Gamma', \mathtt{x} : \mathtt{T}, \Gamma''$
- $\Gamma(\mathtt{C}) = \mathtt{C}\ \mathsf{ext}\ \mathtt{C'}\ \mathsf{impl}\ \mathtt{I_1}, \ldots \mathtt{I_n}\{\mathtt{v_1} : \mathtt{T_1}, \ldots \mathtt{v_m} : \mathtt{T_m}, \mathtt{m_1} : \mathtt{MT_1}, \ldots \mathtt{m_k} : \mathtt{MT_k}\}$ *iff* $\Gamma = \Gamma', \mathtt{C}\ \mathsf{ext}\ \mathtt{C'}\ \mathsf{impl}\ \mathtt{I_1}, \ldots \mathtt{I_n}\{\mathtt{v_1} : \mathtt{T_1}, \ldots \mathtt{v_m} : \mathtt{T_m}, \mathtt{m_1} : \mathtt{MT_1}, \ldots \mathtt{m_k} : \mathtt{MT_k}\}, \Gamma''$
- $\Gamma(\mathtt{I}) = \mathtt{I}\ \mathsf{ext}\ \mathtt{I_1}, \ldots \mathtt{I_n}\{\mathtt{m_1} : \mathtt{MT_1}, \ldots \mathtt{m_k} : \mathtt{MT_k}\}$ *iff* $\Gamma = \Gamma'', \mathtt{I}\ \mathsf{ext}\ \mathtt{I_1}, \ldots \mathtt{I_n}\{\mathtt{m_1} : \mathtt{MT_1}, \ldots \mathtt{m_k} : \mathtt{MT_k}\}, \Gamma''$
- $\Gamma(\mathtt{I}) = \mathsf{Undef}$ *otherwise*

The chapters 8.2 and 9 in [13] describe restrictions imposed on component (*i.e.* variable or method) definitions in a class or interface. We first introduce some functions to find the class components:

- $FDec(\Gamma, \mathtt{C}, \mathtt{v})$ indicates the nearest superclass of `C` (possibly `C` itself) which contains a declaration of the instance variable `v` and its declared type;
- $FDecs(\Gamma, \mathtt{C}, \mathtt{v})$ indicates all the field declarations for `v`, which were declared in a superclass of `C`, and possibly hidden by `C`, or another superclass.

- $MDecs(\Gamma, \mathtt{C}, \mathtt{m})$ indicates all method declarations (*i.e.* both the class of the declaration and the signature) for method $\mathtt{m}$ in class $\mathtt{C}$, or inherited from one of its superclasses, and not hidden by any of its superclasses;
- $MSigs(\Gamma, \mathtt{C}, \mathtt{m})$ returns all signatures for method $\mathtt{m}$ in class $\mathtt{C}$, or inherited and not hidden by any of its superclasses.

An example can be found in section 6.

Definition 3 *For an environment Γ, containing a class declaration for $\mathtt{C}$, i.e. $\Gamma = \Gamma', \mathtt{C}\ \mathsf{ext}\ \mathtt{C}'\ \mathsf{imp}|\ \mathtt{I}_1, \ldots \mathtt{I}_n\{\mathtt{v}_1 : \mathtt{T}_1, \ldots \mathtt{v}_k : \mathtt{T}_k,\ \mathtt{m}_1 : \mathtt{MT}_1, \ldots \mathtt{m}_l : \mathtt{MT}_l\}$, Γ'', we define:*

- $FDec(\Gamma, \mathtt{Object}, \mathtt{v}) = \mathsf{Undef}$ *for any* $\mathtt{v}$
  $FDec(\Gamma, \mathtt{C}, \mathtt{v}) = (\mathtt{C}, \mathtt{T}_j)$ *iff* $\mathtt{v} = \mathtt{v}_j$
  $FDec(\Gamma, \mathtt{C}, \mathtt{v}) = FDec(\Gamma, \mathtt{C}', \mathtt{v})$ *iff* $\mathtt{v} \neq \mathtt{v}_j\ \forall \mathtt{j} \in \{1, \ldots \mathtt{k}\}$
- $FDecs(\Gamma, \mathtt{C}, \mathtt{v}) = \{(\mathtt{C}', \mathtt{T}) \quad | \quad \Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}', (\mathtt{C}', \mathtt{T}) = FDec(\Gamma, \mathtt{C}', \mathtt{v})\}$
- $MDecs(\Gamma, \mathtt{Object}, \mathtt{m}) = \emptyset$
  $MDecs(\Gamma, \mathtt{C}, \mathtt{m}) = \{\ (\mathtt{C}, \mathtt{MT}_j) \quad | \quad \mathtt{m} = \mathtt{m}_j\ \}$
  $\cup\{\ (\mathtt{C}'', \mathtt{MT}'') \quad | \quad (\mathtt{C}'', \mathtt{MT}'') \in MDecs(\Gamma, \mathtt{C}', \mathtt{m}), \quad$ *and*
  $\forall \mathtt{j} \in \{1, \ldots \mathtt{l}\} :$ *if* $\mathtt{m} = \mathtt{m}_j$ *then* $Args(\mathtt{MT}_j) \neq Args(\mathtt{MT}'')\ \}$
- $MSigs(\Gamma, \mathtt{C}, \mathtt{m}) = \{\ \mathtt{MT} \quad | \quad \exists \mathtt{C}''$ *with* $(\mathtt{C}'', \mathtt{MT}) \in MDecs(\Gamma, \mathtt{C}, \mathtt{m})\ \}$

Similar to classes, we introduce the following functions to look up the interface components: $MDecs(\Gamma, \mathtt{I}, \mathtt{m})$ is all the method declarations (*i.e.* the interface of the declaration and the signature) for method $\mathtt{m}$ in interface $\mathtt{I}$, or inherited – and not hidden – from any of its superinterfaces; $MSigs(\Gamma, \mathtt{I}, \mathtt{m})$ returns all signatures for method $\mathtt{m}$ in interface $\mathtt{I}$, or inherited – and not hidden – from a superinterface.

Definition 4 *For an environment Γ, containing an interface declaration for $\mathtt{I}$, i.e. $\Gamma = \Gamma', \mathtt{I}\ \mathsf{ext}\ \mathtt{I}_1, \ldots \mathtt{I}_n\{\mathtt{m}_1 : \mathtt{MT}_1, \ldots \mathtt{m}_k : \mathtt{MT}_k\}, \Gamma''$ we define:*

- $MDecs(\Gamma, \mathtt{I}, \mathtt{m}) = \{\ (\mathtt{I}, \mathtt{MT}_j) \quad | \quad \mathtt{m} = \mathtt{m}_j\ \} \cup \{\ (\mathtt{I}', \mathtt{MT}') \quad |$
  $\exists \mathtt{j} \in \{1, \ldots \mathtt{n}\}$ *with* $(\mathtt{I}', \mathtt{MT}') \in MDecs(\Gamma, \mathtt{I}_j, \mathtt{m})$
  and $\forall \mathtt{i} \in \{1, \ldots \mathtt{k}\}$ *if* $\mathtt{m} = \mathtt{m}_i$ *then* $Args(\mathtt{MT}') \neq Args(\mathtt{MT}_i)\ \}$
- $MSigs(\Gamma, \mathtt{I}, \mathtt{m}) = \{\ \mathtt{MT}' \quad | \quad \exists \mathtt{I}' : (\mathtt{I}', \mathtt{MT}') \in MDecs(\Gamma, \mathtt{I}, \mathtt{m})\ \}$

The following lemma says that if a type $\mathtt{T}$ inherits a method signature from another type $\mathtt{T}'$ *i.e.* if $(\mathtt{T}', \mathtt{MT}) \in MDecs(\Gamma, \mathtt{T}, \mathtt{m})$, then $\mathtt{T}'$ is either a class or an interface exporting that method, and no other superclass of $\mathtt{T}$, which is a subclass of $\mathtt{T}'$ exports a method with the same identifier and argument types. Also, if a class $\mathtt{C}$ inherits a field declaration for $\mathtt{v}$, then there exists a $\mathtt{C}'$, a superclass of $\mathtt{C}$ which contains the declaration of $\mathtt{v}$.

Lemma 1. *For any environment Γ, type $\mathtt{T}$, $\mathtt{T}'$ and identifiers $\mathtt{v}$ and $\mathtt{m}$:*

- $(\mathtt{T}', \mathtt{MT}) \in MDecs(\Gamma, \mathtt{T}, \mathtt{m}) \implies$

- $\Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{T}'$ *and* $\Gamma(\mathtt{T}') = \mathtt{T}'$ ext ...impl ...$\{\ldots \mathtt{m} : \mathtt{MT} \ldots\}$ *and*
  $\forall \mathtt{C} \neq \mathtt{T}', \mathtt{T}''$ *with:* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{T}'$, $\Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{C}$:
  $\Gamma(\mathtt{C}) \neq \mathtt{C}$ ext ...impl$\ldots\{\ldots \mathtt{m} : Args(\mathtt{MT}) \rightarrow \mathtt{T}''\}$
  or
- $\Gamma \vdash \mathtt{T} \leq \mathtt{T}'$ *and* $\Gamma(\mathtt{T}') = \mathtt{T}'$ ext $\ldots\{\ldots \mathtt{m} : \mathtt{MT} \ldots\}$ *and*
  $\forall \mathtt{I} \neq \mathtt{I}', \mathtt{T}''$: *with* $\Gamma \vdash \mathtt{I} \leq \mathtt{T}', \Gamma \vdash \mathtt{T} \leq \mathtt{I}$:
  $\Gamma(\mathtt{I}) \neq \mathtt{I}$ ext $\ldots\{\ldots \mathtt{m} : Args(\mathtt{MT}) \rightarrow \mathtt{T}''\}$

– $FDec(\Gamma, \mathtt{C}, \mathtt{v}) = (\mathtt{C}', \mathtt{T}') \implies \Gamma(\mathtt{C}') = \mathtt{C}' \ldots\{\ldots \mathtt{v} : \mathtt{T} \ldots\}$ *and* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}'$
and $\forall \mathtt{C}'', \mathtt{T}''$ *with* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}'', \Gamma \vdash \mathtt{C}'' \sqsubseteq \mathtt{C}' : \Gamma(\mathtt{C}'') \neq \mathtt{C}''$ ext ...impl...$\{\ldots \mathtt{v} : \mathtt{T}''\}$

When a new class is declared as $\mathtt{C}$ ext $\mathtt{C}'$ impl $\mathtt{I}_1, \ldots \mathtt{I}_n\{\mathtt{v}_1 : \mathtt{T}_1, \ldots \mathtt{v}_k : \mathtt{T}_k, \mathtt{m}_1 : \mathtt{MT}_1, \ldots \mathtt{m}_l : \mathtt{T}_l\}$, [13] imposes the following requirements:

- there can be sequences of superinterfaces, instance variable declarations, and instance method declarations;
- the previous declarations are well-formed;
- there is no prior declaration of $\mathtt{C}$
- there are no cyclic subclass dependencies between $\mathtt{C}'$ and $\mathtt{C}$
- the declarations of the class $\mathtt{C}'$, interfaces $\mathtt{I}_j$ and variable types $\mathtt{T}_j$ may precede or *follow* the declaration for $\mathtt{C}$ – this is why we require $\Gamma \vdash \mathtt{C}' \sqsubseteq \mathtt{C}'$, rather than $\Gamma' \vdash \mathtt{C}' \sqsubseteq \mathtt{C}'$;
- the $\mathtt{MT}_j$ are method types;
- instance variable identifiers are unique;
- instance methods with the same identifier must have different argument types;
- a method overriding an inherited method must have a result type that *widens* to the result type of the overridden method – here we follow [17] instead of [13] which requires the result types to be identical; we prefer the former because it is a more general definition;
- "unless a class is abstract, the declarations of methods defined in each direct superinterface must be implemented either by a declaration in this class, or by an existing method declaration inherited from a superclass" - again we follow [17] instead of [13], and we require the implementing method to have a result type that *widens* to the result type of the interfaces method, instead of requiring them to be identical.

When a new interface $\mathtt{I}$ is introduced as $\mathtt{I}$ ext $\mathtt{I}_1, \ldots \mathtt{I}_n\{\ \mathtt{m}_1 : \mathtt{MT}_1, \ldots \mathtt{m}_l : \mathtt{T}_l\}$, the following requirements must be satisfied:

- there may be sequences of superinterfaces and instance method declarations;
- the previous declarations are well-formed;
- there is no prior declaration of $\mathtt{I}$;
- there are no cyclic subinterface dependencies between $\mathtt{I}$ and $\mathtt{I}_j$;
- the $\mathtt{I}_j$ are interfaces – whose declaration may precede or *follow* that of $\mathtt{I}$;
- the $\mathtt{MT}_j$ are method types;
- instance methods with the same identifier must have different argument types;

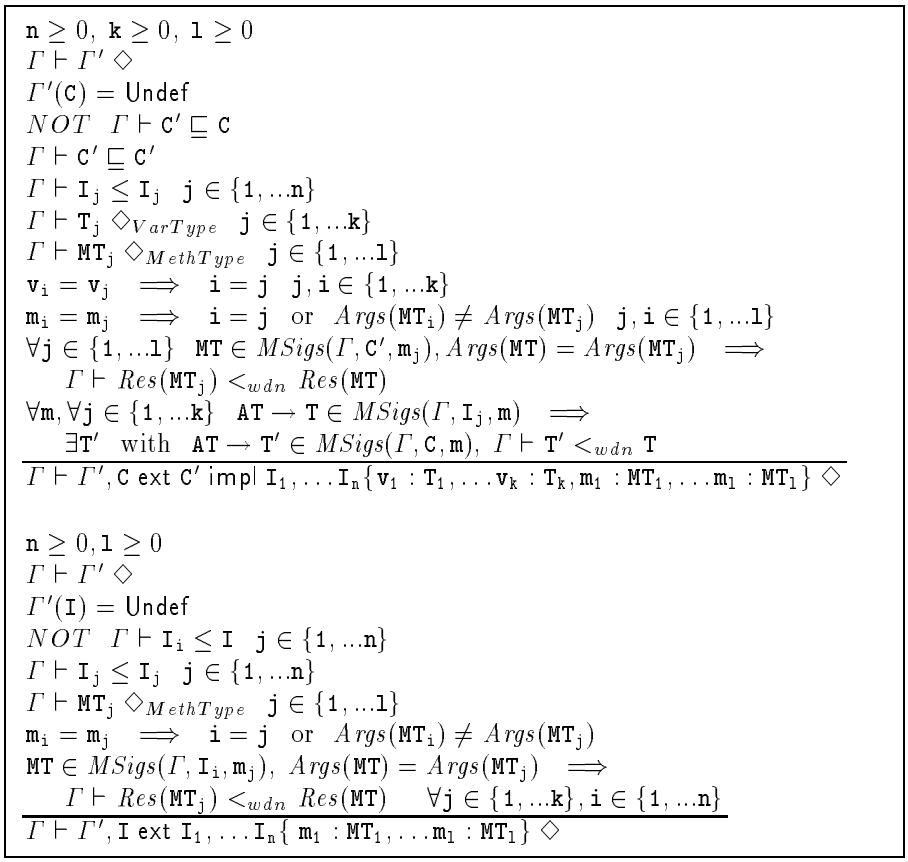

Fig. 7. class and interface declarations

– a method overriding an inherited method (a method is inherited if defined in one of the superinterfaces, and it is overridden if it has the same identifier and same argument types) must have a result type that widens to the result type of the overridden method – as for classes, here too we follow [17] instead of [13].

2.4 Properties of well-formed environments

Lemma 2. *If $\Gamma \vdash \Gamma\ \diamond$, then Γ contains at most one declaration for any identifier, and there are no cycles in the $\sqsubseteq$ and $\leq$ relationship.*

In the following lemma we show that two types that are in the subclass relationship are classes, that $\sqsubseteq$ is reflexive, transitive and antisymmetric, that the

subclass hierarchy forms a tree, that two types that are in the subinterface relationship are interfaces, and that $\leq$ is transitive, reflexive and antisymmetric. Note, that unlike $\sqsubseteq$, $\leq$ does not form a tree:

Lemma 3. *If* $\Gamma \vdash \Gamma\ \diamond$, *then:*

- $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}' \implies \Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}$ *and* $\Gamma \vdash \mathtt{C}' \sqsubseteq \mathtt{C}'$
- $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}'$ *and* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}'' \implies \Gamma \vdash \mathtt{C}' \sqsubseteq \mathtt{C}''$ *or* $\Gamma \vdash \mathtt{C}'' \sqsubseteq \mathtt{C}'$
- *The* $\sqsubseteq$ *relationship is a partial order.*
- $\Gamma \vdash \mathtt{I} \leq \mathtt{I}' \implies \Gamma \vdash \mathtt{I}' \leq \mathtt{I}'$ *and* $\Gamma \vdash \mathtt{I} \leq \mathtt{I}$
- *The* $\leq$ *relationship is a partial order.*

The following lemma says that widening is reflexive, transitive and antisymmetric; that if an interface widens to another type, then the second type is a superinterface of the first; that if a type widens to a class, then the type is a subclass of that class; that if a class widens to an interface `I`, then the class implements a subinterface of `I`; that if an interface widens to another type, then the interface is identical to the type, or one of its immediate superinterfaces is a subinterface of that type.

Lemma 4. *If* $\Gamma \vdash \Gamma\ \diamond$, *then:*

- $\Gamma \vdash \mathtt{I} \leq \mathtt{I}$ *and* $\Gamma \vdash \mathtt{I} <_{wdn} \mathtt{T} \implies \Gamma \vdash \mathtt{I} \leq \mathtt{T}$
- $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}$ *and* $\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{C} \implies \Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{C}$
- $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}$ *and* $\Gamma \vdash \mathtt{C} <_{wdn} \mathtt{I}$ *and* $\Gamma \vdash \mathtt{I} \leq \mathtt{I} \implies$
  $\exists \mathtt{C}', \mathtt{I}'$: $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}'$, $\Gamma \vdash \mathtt{C}' :_{imp} \mathtt{I}'$ *and* $\Gamma \vdash \mathtt{I}' \leq \mathtt{I}$
- $\Gamma = \Gamma'$, $\mathtt{I}\ \mathtt{ext}\ \mathtt{I}_1 \ldots \mathtt{I}_n\{\ldots\}, \Gamma''$, *and* $\Gamma \vdash \mathtt{I} <_{wdn} \mathtt{T} \implies$
  $\mathtt{I} = \mathtt{T}$ *or* $\Gamma \vdash \mathtt{I}_k \leq \mathtt{T}$ *for a* $\mathtt{k} \in \{1, \ldots \mathtt{n}\}$
- *The* $<_{wdn}$ *relationship is a partial order.*

If a type `T` widens to another type `T'`, and `T'` has a method `m`, then there exists in `T` a unique method `m` with the same argument types, and whose return type can be widened to that of `T'`. Note that we follow the more general rule from [17] as opposed to [13].

Lemma 5. *If* $\Gamma \vdash \Gamma\ \diamond$, *for types* `T` *and* `T'`, *with* $\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{T}'$, *and* $\mathtt{MT}' \in MSigs(\Gamma, \mathtt{T}', \mathtt{m})$:

$\exists_1 \mathtt{MT} \in MSigs(\Gamma, \mathtt{T}, \mathtt{m})$ *with* $Args(\mathtt{MT}) = Args(\mathtt{MT}')$.
Furthermore, $\Gamma \vdash Res(\mathtt{MT}) <_{wdn} Res(\mathtt{MT}')$

From now on we assume implicitly that all environments are well-formed.

3 The type rules

Type checking is described in terms of a type inference system. In parallel with type checking the program is slightly modified, and enriched with type information. The Java$_s$-program is turned into a Java$_{se}$-program. The enriching of the program by type information is described by the mapping *Comp*:

$$Comp : \text{Java}_s \longrightarrow \text{Java}_{se}$$

3.1 Java$_{se}$, enriching Java$_s$

Some compile-time type information is necessary for the execution of Java method calls and of instance variable access. This information is calculated when type checking, and needs to be available during execution.
Therefore, we defined Java$_{se}$, an extended version of Java$_s$, which includes the appropriate type information. Furthermore, terms like $\alpha_{\mathtt{i}}$ represent references to objects, which will be necessary for describing the operational semantics. Also, in order to describe method evaluation without using closures, in Java$_s$ we allow an expression to consist of a sequence of statements. Finally, execution of Java$_{se}$ programs may raise the exceptions NullPointExc, indicating an attempt to access an instance variable of the null pointer, ArrStoreExc indicating an attempt to assign a value of the wrong class to an array component, IndOutBndExc indicating an index out of the array bounds, and NegSzeExc, when attempting to create a new array value of a negative size. The syntax of Java$_{se}$ may be obtained from the syntax of Java$_s$ by applying the modifications and additions in figure 8:

Expr	::=	...	
	|	*Expr.*[*ArgType*]MethName(*Expr**)	instead of *Expr.*MethName(*Expr**)
	|	*Stmts*	
Var	::=	...	
	|	*Var.*[ClassName]VarName	instead of *Var.*VarName
	|	$\alpha_{\mathtt{i}}$	i *an integer*
Value	::=	... | *RefValue* | *Exception*	
RefValue	::=	$\alpha_{\mathtt{i}}$	i *an integer*
Exception	::=	NullPointExc | ArrStoreExc	
	|	IndOutBndExc | NegSzeExc	

Fig. 8. type rules for Java$_{se}$

3.2 Types for Java$_s$

The types for variables, primitive values and null are described in figure 9.
The type rules for assignments, return statements, statement sequences and conditionals are given in figure 10. An expression of type T′ can be assigned to a variable of a type T, if T′ can be widened to T. A statement sequence has the same type as its last statement. A return statement has void type, or the same type as the expression it returns. A conditional consists of two statement sequences of the same type.

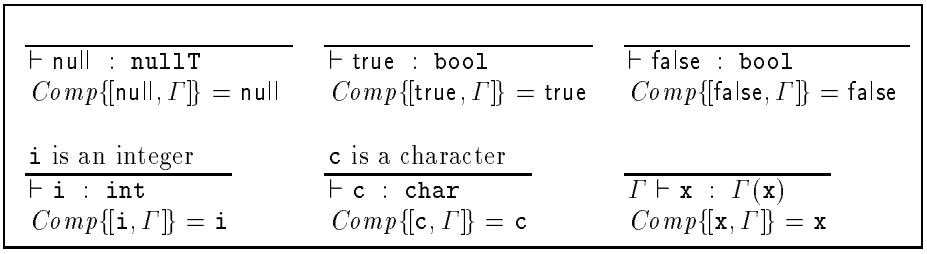

Fig. 9. types of primitive values and variables

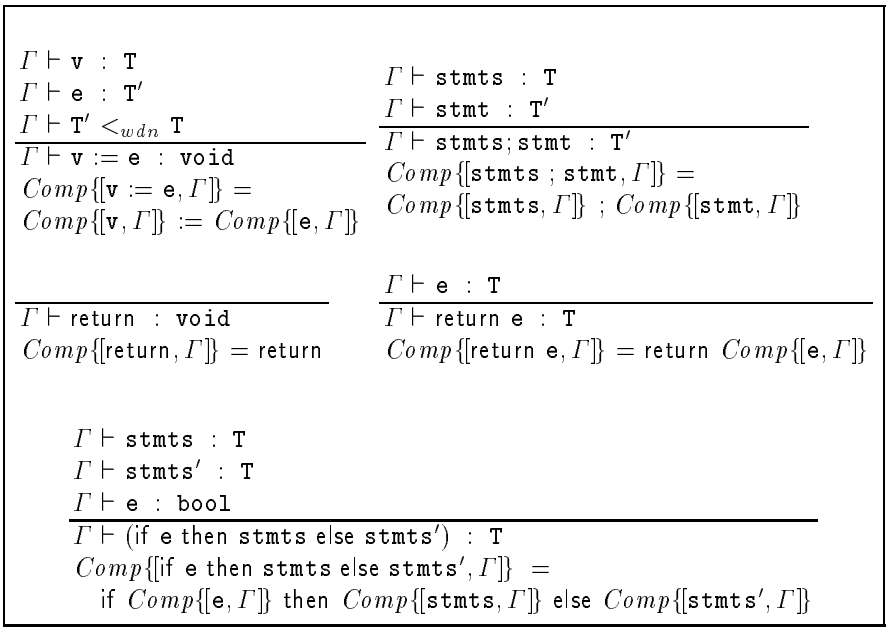

Fig. 10. types of statements

Figure 11 contains the type rules for newly created objects or arrays. For a class `C`, the expression `new C` has type `C`. For a simple type `T`, the expression `new T`$[e_1]\ldots[e_n][]_1\ldots[]_k$ is a `n+k`-dimensional array of elements of type `T`.
Figure 12 contains the type rules for array and field accesses. The possibility of a runtime exception is described with the operational semantics in figures 20 and 18. Only classes have fields.
Figure 13 contains the type rules for method bodies and method calls, as in ch. 15.11, [17]: A method is *applicable* if the actual parameter types can be widened to the corresponding formal parameter types. A signature is *more special* than another signature, if and only if it is defined in a subclass or subinterface and all argument types can be widened to from the argument types of the second

$$\frac{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}}{\begin{array}{l}\Gamma \vdash \mathtt{new\ C\ :\ C} \\ Comp\{[\mathtt{new\ C}, \Gamma]\} = \mathtt{new\ C}\end{array}} \qquad \frac{\begin{array}{l}\mathtt{n} \geq 0, \mathtt{k} \geq 0, \mathtt{n+k} \geq 1 \\ \Gamma \vdash \mathtt{T} \diamond_{VarType},\ \text{a simple type} \\ \Gamma \vdash \mathtt{e_i\ :\ int} \quad \mathtt{i} \in \{\mathtt{1}, ...\mathtt{n}\}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{new\ T[e_1]} \ldots \mathtt{[e_n][]_1} \ldots \mathtt{[]_k\ :\ T[]_1} \ldots \mathtt{[]_{n+k}} \\ Comp\{[\mathtt{new\ T[e_1]} \ldots \mathtt{[e_n][]_1} \ldots \mathtt{[]_k}, \Gamma]\} = \\ \quad \mathtt{new\ T}[Comp\{[\mathtt{e_1}, \Gamma]\}] \ldots [Comp\{[\mathtt{e_n}, \Gamma]\}]\mathtt{[]_1} \ldots \mathtt{[]_k}\end{array}}$$

Fig. 11. object and array creation type rules

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{v\ :\ T[]} \\ \Gamma \vdash \mathtt{e\ :\ int}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{v[e]\ :\ T} \\ Comp\{[\mathtt{v[e]}, \Gamma]\} = \\ \quad Comp\{[\mathtt{v}, \Gamma]\}[Comp\{[\mathtt{exp}, \Gamma]\}]\end{array}} \qquad \frac{\begin{array}{l}\Gamma \vdash \mathtt{v\ :\ T} \\ FDec(\Gamma, \mathtt{T}, \mathtt{f}) = (\mathtt{C}, \mathtt{T'})\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{v.f\ :\ T'} \\ Comp\{[\mathtt{v.f}, \Gamma]\} = Comp\{[\mathtt{v}, \Gamma]\}.[\mathtt{C}]\mathtt{f}\end{array}}$$

Fig. 12. array and field access type rules

signature; this defines a partial order. The most special signatures are the minima of the "more special" partial order.

Definition 5 *For an environment* Γ*, variable types* $\mathtt{T}$ *and* $\mathtt{T_i}$*,* $\mathtt{i} \in \{\mathtt{1}, ...\mathtt{n+1}\}$*, and identifier* $\mathtt{m}$*, the* most special declarations *are defined as follows:*

- $ApplMeths(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T_1} \times \ldots \times \mathtt{T_n}) = \{(\mathtt{T'}, \mathtt{MT'}) \mid (\mathtt{T'}, \mathtt{MT'}) \in MDecs(\Gamma, \mathtt{T}, \mathtt{m})$ *and* $\mathtt{MT'} = \mathtt{T'_1} \times \ldots \times \mathtt{T'_n} \rightarrow \mathtt{T'_{n+1}}$ *and* $\Gamma \vdash \mathtt{T_i} <_{wdn} \mathtt{T'_i}$ *for* $\mathtt{i} \in \{\mathtt{1}, ...\mathtt{n}\}\}$
- $(\mathtt{T}, \mathtt{T_1} \times \ldots \times \mathtt{T_n} \rightarrow \mathtt{T_{n+1}})$ *is* more special than $(\mathtt{T'}, \mathtt{T'_1} \times \ldots \times \mathtt{T'_n} \rightarrow \mathtt{T'_{n+1}})$ *iff* $\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{T'}$ *and* $\Gamma \vdash \mathtt{T_i} <_{wdn} \mathtt{T'_i}$ *for all* $\mathtt{i} \in \{\mathtt{1}, ...\mathtt{n}\}$
- $MostSpec(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T_1} \times \ldots \times \mathtt{T_n}) =$ $\{(\mathtt{T'}, \mathtt{MT'}) \mid (\mathtt{T'}, \mathtt{MT'}) \in ApplMeths(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T_1} \times \ldots \times \mathtt{T_n})$ *and if* $(\mathtt{T''}, \mathtt{MT''}) \in ApplMeths(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T_1} \times \ldots \times \mathtt{T_n})$ *and* $(\mathtt{T''}, \mathtt{MT''})$ *is* more special than $(\mathtt{T'}, \mathtt{MT'})$ *then* $\mathtt{T''} = \mathtt{T'}$ *and* $\mathtt{MT'} = \mathtt{MT''}\}$

The signatures of the more specific applicable methods are contained in the set $MostSpec(,,,)$. A message expression is type correct when this set contains exactly one pair. The argument types of the signature of this pair is stored as the *method descriptor*, *c.f.* ch.15.5 in [13], and the result type of the signature is the type of the message expression.
The renaming of the variables in the method body (*i.e.* $\mathtt{stmts}[\mathtt{z_1}/\mathtt{x_1}, \ldots, \mathtt{z_n}/\mathtt{x_n}]$) is necessary in order to avoid name clashes and also, in order for the lemma 9

$$\frac{\Gamma \vdash \mathtt{e_i} \ : \ \mathtt{T_i} \quad \mathtt{i} \in \{1,...\mathtt{n}\}, \mathtt{n} \geq 1 \qquad \mathit{MostSpec}(\Gamma, \mathtt{m}, \mathtt{T_1}, \mathtt{T_2} \times ... \times \mathtt{T_n}) \ = \{(\mathtt{T}, \mathtt{MT})\}}{\begin{array}{l}\Gamma \vdash \mathtt{e_1.m(e_2...e_n)} \ : \ \mathit{Res}(\mathtt{MT}) \\ \mathit{Comp}\{[\mathtt{e_1.m(e_2...e_n)}, \Gamma]\} = \\ \quad \mathit{Comp}\{[\mathtt{e_1}, \Gamma]\}.[\mathit{Args}(\mathtt{MT})]\mathtt{m}(\mathit{Comp}\{[\mathtt{e_2}, \Gamma]\}...\mathit{Comp}\{[\mathtt{e_n}, \Gamma]\})\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{mBody} = \mathtt{m\ is}\ \lambda \mathtt{x_1} : \mathtt{T_1} \ldots \lambda \mathtt{x_n} : \mathtt{T_n}.\{\ \mathtt{stmts}\ \} \\ \mathtt{x_i} \neq \mathtt{this} \quad \mathtt{i} \in \{1,...\mathtt{n}\} \\ \mathtt{z_1}, \ldots, \mathtt{z_n} \text{ are new variables in } \Gamma \\ \mathtt{stmts'} = \mathtt{stmts}[\mathtt{z_1}/\mathtt{x_1}, \ldots, \mathtt{z_n}/\mathtt{x_n}] \\ \Gamma, \mathtt{z_1} : \mathtt{T_1} \ldots \mathtt{z_n} : \mathtt{T_n} \vdash \mathtt{stmts'} \ : \ \mathtt{T'} \\ \Gamma \vdash \mathtt{T'} <_{wdn} \mathtt{T}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{mbody} \ : \ \mathtt{T_1} \times \ldots \times \mathtt{T_n} \rightarrow \mathtt{T} \\ \mathit{Comp}\{[\mathtt{mBody}, \Gamma]\} = \ \mathtt{m\ is}\ \lambda \mathtt{x_1} : \mathtt{T_1} \ldots \lambda \mathtt{x_n} : \mathtt{T_n}.\{\mathit{Comp}\{[\mathtt{stmts}, \Gamma]\}\}\end{array}}$$

Fig. 13. types of method calls and bodies

to hold – as pointed out in [20]. Furthermore, it is worth noticing, that the rules describing method bodies do not determine T, the return type of the method; this is taken from the environment Γ, when applying the rule describing class bodies, as in figure 14.

Figure 14 contains the type rules for class bodies and programs. A class body cBody satisfies its declaration, $\Gamma(\mathtt{C})$, if it provides a method body for each of the method declarations contained in $\Gamma(\mathtt{C})$.

Note, that the method bodies $\mathtt{mBody_i}$ are type checked in the environment Γ, this : C, which does not contain the instance variable declarations $\mathtt{v_1} : \mathtt{T_1}$ ..., $\mathtt{v_k} : \mathtt{T_k}$. Thus, by the type system, we force the use of the expression $\mathtt{this.v_j}$ as opposed to $\mathtt{v_j}$.

A program $\mathtt{p} = \{\ \mathtt{cBody_1}, \ldots \mathtt{cBody_n}\ \}$ is well-typed, if it contains a class body for each declared class, and if all class bodies, $\mathtt{cBody_i}$, are well-typed and satisfy their declarations. Furthermore, each class is transformed by *Comp*.

The following two functions will be needed for the operational semantics. The function $\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{cBody})$ finds the method body with identifier m and argument types AT, in the class body cBody – if any exists. It can easily be seen that because of the requirements for classes in 2.3, if the environment Γ is well-formed, the function $\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{cBody})$ returns either an empty set or a set with one element.

Definition 6 *For a class body* $\mathtt{cBody} = \mathtt{C\ ext\ C'}\ \{\ \mathtt{mBody_1}, \ldots \mathtt{mBody_n}\ \}$*, declared in* Γ *as* $\Gamma(\mathtt{C}) = \mathtt{C\ ext\ C'\ impl}\ \ldots \{\ \mathtt{m_1} : \mathtt{MT_1} \ldots \mathtt{m_n} : \mathtt{MT_n}\ \}$*, we define:*
$\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{cBody}) = \{\mathtt{mPS_j} \mid \ \mathtt{mBody_j} = \mathtt{m\ is\ mPS_j}$ *and* $\mathit{Args}(\mathtt{MT_j}) = \mathtt{AT}\}$

The function $\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{C}, \mathtt{p})$ finds the method body with identifier m and argument types AT, in the nearest superclass of class C – if any exists. It returns

$$\frac{\begin{array}{l}\mathtt{n} \geq 0, \mathtt{k} \geq 0, \mathtt{m} \geq 0 \\ \Gamma \vdash \Gamma\ \Diamond \\ \Gamma(\mathtt{C}) = \mathtt{C}\ \mathsf{ext}\ \mathtt{C}'\ \mathsf{imp|}\ \mathtt{I}_1 \ldots \mathtt{I}_n\{\ \mathtt{v}_1 : \mathtt{T}_1 \ldots \mathtt{v}_k : \mathtt{T}_k, \mathtt{m}_1 : \mathtt{MT}_1 \ldots \mathtt{m}_l : \mathtt{MT}_l\ \} \\ \mathtt{cBody} = \mathtt{C}\ \mathsf{ext}\ \mathtt{C}'\ \{\ \mathtt{mBody}_1, \ldots \mathtt{mBody}_l\ \} \\ \Gamma(\mathtt{this}) = \mathsf{Undef} \\ \mathtt{mBody}_i = \mathtt{m}_i\ \mathsf{is}\ \mathtt{mPrsSts}_i \quad \mathtt{i} \in \{1, \ldots \mathtt{l}\} \\ \Gamma, \mathtt{this} : \mathtt{C} \vdash \mathtt{mBody}_i\ :\ \mathtt{MT}_i \quad \mathtt{i} \in \{1, \ldots \mathtt{l}\}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{cBody}\ :\ \Gamma(\mathtt{C}) \\ \mathit{Comp}\{[\mathtt{cBody}, \Gamma]\} = \mathtt{C}\ \mathsf{ext}\ \mathtt{C}'\ \{\mathit{Comp}\{[\mathtt{mBody}_1, \Gamma]\} \ldots \mathit{Comp}\{[\mathtt{mBody}_l, \Gamma]\}\}\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{C}_1 \ldots \mathtt{C}_n \text{ are all the classes defined in } \Gamma \quad \mathtt{n} \geq 0 \\ \mathtt{p} = \{\ \mathtt{cBody}_1, \ldots \mathtt{cBody}_n\ \} \\ \mathtt{cBody}_i = \mathtt{C}_i\ \mathsf{ext}\ \ldots \{\ \ldots\ \} \quad \text{for } \mathtt{i} \in \{1, \ldots \mathtt{n}\} \\ \Gamma \vdash \mathtt{cBody}_i\ :\ \Gamma(\mathtt{C}_i) \quad \mathtt{i} \in \{1, \ldots \mathtt{n}\}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{p}\ :\ \Gamma \\ \mathit{Comp}\{[\mathtt{p}, \Gamma]\} = \{\ \mathit{Comp}\{[\mathtt{cBody}_1, \Gamma]\} \ldots \mathit{Comp}\{[\mathtt{cBody}_n, \Gamma]\}\ \}\end{array}}$$

Fig. 14. type rules for class bodies and programs

a single pair consisting of the class containing the appropriate method body, and the method body itself or the empty set if none exists.

Definition 7 *For a program* $\mathtt{p} = \{\ \mathtt{cBody}_1, \ldots \mathtt{cBody}_n\ \}$*, we define:*

$\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{C}, \mathtt{p}) =$
  let $\mathtt{cBody} = \mathtt{C}\ \mathsf{ext}\ \mathtt{C}'\ \{\ \ldots\ \}$ *in*
  let $\mathtt{mBody} = \mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{cBody})$ *in*
  if $\mathtt{mBody} = \emptyset$ *then*
    if $\mathtt{C}' = \mathtt{Object}$ *then* $\emptyset$
    else $\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{C}', \mathtt{p})$
  else $(\mathtt{C}, \mathtt{mBody})$

3.3 Properties of the Java$_s$ type system

The following lemma says, that in a well-typed Java$_s$ program any class that widens to a superclass or superinterface provides an implementation for each method exported by the superclass or superinterface.

Lemma 6. *For any well-formed environment* Γ*, variable types* $\mathtt{T}$*,* $\mathtt{T}_1$*, ...,*$\mathtt{T}_n$*,* $\mathtt{T}_{n+1}$*, class* $\mathtt{C}$ *and a Java*$_s$ *program* $\mathtt{p}$*, if:*

- $\Gamma \vdash \mathtt{p}\ :\ \Gamma$
- $\Gamma \vdash \mathtt{C} <_{wdn} \mathtt{T}$
- $\mathtt{T}_1 \times \ldots \mathtt{T}_n \rightarrow \mathtt{T}_{n+1} \in \mathit{MSigs}(\Gamma, \mathtt{T}, \mathtt{m})$

then

– $\exists \mathtt{T}'_{n+1}, \mathtt{C}' : (\mathtt{C}', \mathtt{T}_1 \times \ldots \mathtt{T}_n \rightarrow \mathtt{T}'_{n+1}) \in MDecs(\Gamma, \mathtt{C}, \mathtt{m})$, *and*
$\Gamma \vdash \mathtt{T}'_{n+1} <_{wdn} \mathtt{T}_{n+1}$ *and* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}'$ *and*
– $MethBody(\mathtt{m}, \mathtt{T}_1 \times \ldots \mathtt{T}_n, \mathtt{p}, \mathtt{C}) = (\mathtt{C}', \lambda \mathtt{x}_1 : \mathtt{T}_1, \ldots \lambda \mathtt{x}_n : \mathtt{T}_n.\{\ \mathtt{stmts}\ \})$ *and*
$\Gamma, \mathtt{this} : \mathtt{C}', \mathtt{x}_1 : \mathtt{T}_1, \ldots \mathtt{x}_n : \mathtt{T}_n \vdash \mathtt{stmts} : \mathtt{T}''_{n+1}$ *and* $\Gamma \vdash \mathtt{T}''_{n+1} <_{wdn} \mathtt{T}'_{n+1}$

3.4 Absence of the subsumption rule

The type inference system described in the previous sections does not have a subsumption rule. The *subsumption rule* says, that any expression of type $\mathtt{T}$, also has type $\mathtt{T}'$ if $\mathtt{T}$ is a subtype of $\mathtt{T}'$. In the case of Java, where subtypes are expressed by the $<_{wdn}$ relation, it would have had the form:

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \Gamma \vdash \mathtt{T} <_{wdn} \mathtt{T}'}{\Gamma \vdash \mathtt{e} : \mathtt{T}'}$$

For example, in section 6, the type of `aPhil.like` is `Phil`, but the type of `pascal.like` is `Food`, although $\Gamma_0 \vdash \mathtt{aPhil} : \mathtt{Phil}$, $\Gamma_0 \vdash \mathtt{pascal} : \mathtt{FrPhil}$, and $\Gamma_0 \vdash \mathtt{FrPhil} <_{wdn} \mathtt{Phil}$. In fact, introduction of the subsumption rule would make this type system non-deterministic – although [6] develops a system for Java which has a subsumption rule, and in which the types of method call and field access are determined by using the *minimal types* of the expressions.

3.5 Extending the type rules to Java$_{se}$

The Java$_{se}$ syntax is in most parts identical to that of Java$_s$. For these cases the type rules are identical. The only cases where the syntax differs are method call, field access, and the object references $\alpha_{\mathtt{i}}$. These are shown in figure 15.
The type of a reference depends on the class of the object pointed at in the current state σ (states will be introduced in section 4), therefore, the type of a Java$_{se}$ term depends on both the environment *and* the state, and type assertions for Java$_{se}$ terms $\mathtt{t}$ have the form $\Gamma, \sigma \vdash \mathtt{t} : \mathtt{T}$.
If an object is stored at address $\alpha_{\mathtt{i}}$, then its class is the type of the reference $\alpha_{\mathtt{i}}$. If a $\mathtt{k}$-dimensional array of $\mathtt{T}$ is stored at $\alpha_{\mathtt{i}}$, then the $\mathtt{k}$-dimensional array of $\mathtt{T}$, $\mathtt{T}[]_1 \ldots []_k$ is the type of this reference. Objects and array values are defined in section 4.
The difference between the type of a field access expression in Java$_s$ and Java$_{se}$ is, that in Java$_{se}$ the type depends on the descriptor (*i.e.* $\mathtt{C}$) instead of the type of the variable at the left of the field access (*i.e.* $\mathtt{T}$).
In Java$_{se}$ method calls we search for appropriate methods, using the descriptor signature ($\mathtt{T}_2 \times \ldots \times \mathtt{T}_n$), instead of the types of the actual expressions ($\mathtt{T}'_2, \ldots \mathtt{T}'_n$). For this search we first examine the class of the receiver expression for a method body with appropriate argument types, and then *its* superclasses:

Definition 8 *Given environment Γ, types $\mathtt{T}_1, \ldots \mathtt{T}_n$, argument types $\mathtt{AT}=\mathtt{T}_2 \times \ldots \times \mathtt{T}_n$ and an identifier $\mathtt{m}$, we define:*

$$\frac{\sigma(\alpha_i) = << \ldots >>^{C}}{\Gamma, \sigma \vdash \alpha_i \ : \ C} \qquad \frac{\sigma(\alpha_i) = << \ldots >>^{T[n_1]\ldots[n_k]}}{\Gamma, \sigma \vdash \alpha_i \ : \ T[]_1 \ldots []_k}$$

$$\frac{\Gamma, \sigma \vdash v \ : \ T \quad \Gamma, \sigma \vdash T <_{wdn} C \quad FDec(\Gamma, C, f) = (C, T')}{\Gamma, \sigma \vdash v.[C]f \ : \ T'}$$

$$\frac{\Gamma, \sigma \vdash e_i \ : \ T'_i \quad i \in \{1, \ldots n\}, n \geq 0 \quad \Gamma, \sigma \vdash T'_i <_{wdn} T_i \quad i \in \{2, \ldots n\} \quad FirstFit(\Gamma, m, T'_1, T_2 \times \ldots \times T_n) = \{(T, MT)\}}{\Gamma, \sigma \vdash e_1.[T_2 \times \ldots \times T_n]m(e_2 \ldots e_n) \ : \ Res(MT)}$$

Fig. 15. types for Java$_{se}$

$$FirstFit(\Gamma, m, T_1, AT) = \{(T, MT) \mid (T, MT) \in MDecs(\Gamma, T_1, m) \text{ and } Args(MT) = AT\}$$

Lemma 7. *For a well-formed environment Γ, types T'_1, T_1 $\ldots T_n$, argument types, $AT = T_2 \times \ldots \times T_n$, where $\Gamma \vdash T_1 <_{wdn} T'_1$:*

- *the set $FirstFit(\Gamma, m, T_1, AT)$ contains up to one element*
- $\exists T', MT' : FirstFit(\Gamma, m, T'_1, AT) = (T', MT') \implies$
  $\exists T, MT : FirstFit(\Gamma, m, T_1, AT) = (T, MT)$ *and*
  $\Gamma \vdash T <_{wdn} T'$ *and* $\Gamma \vdash Res(MT) <_{wdn} Res(MT')$

3.6 Properties of the Java$_{se}$ type system

We expect the type of a Java$_{se}$-expression to be related to the type of the original Java$_s$-expression. In fact, they are identical. The type system assigns unique types to any well-typed Java$_s$ or Java$_{se}$ term.

Lemma 8. *For types T, T', state σ, environment Γ, Java$_s$ term t, and Java$_{se}$ term t':*

- $\Gamma \vdash t \ : \ T \implies \Gamma, \sigma \vdash Comp\{[t, \Gamma]\} \ : \ T$
- $\Gamma \vdash t \ : \ T$ *and* $\Gamma \vdash t \ : \ T' \implies T = T'$.
- $\Gamma, \sigma \vdash t' \ : \ T$ *and* $\Gamma, \sigma \vdash t' \ : \ T' \implies T = T'$.
- *Expressions containing exceptions are not type correct.*

4 The operational semantics

Figure 16 describes the run time model for the operational semantics.
Firstly, we need a notion of state. The state is flat; it consists of mappings from identifiers to primitive values or to references, and from references to objects or arrays.
Every object is annotated by its class. An object consists of a sequence of labels and values. Each label also carries the class in which it was defined; this is needed for labels shadowing labels from superclasses, cf [13] ch. 9.5. For example, as in section 6, $<<$ `like Phil:` α_5, `like FrPhil: croissant` $>>^{FrPhil}$ is an object

of class `FrPhil`. It inherits the field `like` from `Phil`, and has the field `like` from `FrPhil`.
Arrays carry their dimension and type information, and they consist of a sequence of values for the first dimension. For example $<<3,5,8,11>>^{\texttt{int[]}}$, is a one dimensional array of integers.
Configurations are tuples of Java$_{se}$ terms and states, or just states. The operational semantics is a mapping from programs and configurations to configurations. For a given program `p`, the operational semantics maps configurations to new configurations.

State	::=	(`Ident` $\longrightarrow$ (*Value*))$^*\cup$ (*RefValue* $\longrightarrow$ *ObjectOrArray*)*
ObjectOrArray	::=	*Object* | *Array*
Object	::=	`<<` (`LabelName` `ClassName` : *Value*)*`>>`$^{\texttt{ClassName}}$
Array	::=	`<<` (*Value*)* `>>`^{ArrayType}
Configuration	::=	`<` Java$_{se}$-term, *state* `>` $\cup$ `<` *state* `>`
$\rightsquigarrow$	:	Java$_{se}$program $\longrightarrow$ *Configuration* $\longrightarrow$ *Configuration*
$\rightsquigarrow_{\texttt{p}}$	:	*Configuration* $\longrightarrow$ *Configuration*

Fig. 16. Java$_{se}$ runtime model

Next, we define some operations on states and objects.

4.1 State and object modifications, ground terms

We require objects to be constructed according to their class, array values to conform to their dimension and to consist of values of appropriate types, and variables to contain values of the appropriate type.

Definition 9 *A* value `val` weakly conforms to a type `T` *in an environment* Γ *and a state* σ *iff:*

- *if* `val` *is a primitive value, then* `T` *is a primitive type, and* `val`$\in$`T`*;*
- *if* `val`=`null`*, then* `T` *is a class, interface or array type;*
- *if* `val`$=\alpha_{\texttt{j}}$ *and* `T` *is a class or interface, then there exists a class* `C` *with:* $\Gamma \vdash \texttt{C} <_{wdn} \texttt{T}$*, and* $\sigma(\alpha_{\texttt{j}}) = <<\ldots>>^{\texttt{C}}$*;*
- *if* `val`$=\alpha_{\texttt{j}}$ *and* $\texttt{T} = \texttt{T}'[]_1 \ldots []_k$ *for a simple type* `T`$'$ *and* $\texttt{k} \geq 1$*, then there exists integer* `n` *and type* `T`$''$ *such that :*
  $\Gamma \vdash \texttt{T}'' <_{wdn} \texttt{T}'$, $\sigma(\alpha_{\texttt{j}}) = <<\texttt{val}_0, \ldots \texttt{val}_{\texttt{n}-1}>>^{\texttt{T}''[]_1 \ldots []_{\texttt{k}}}$.

A value `val` conforms to a type `T` *in an environment* Γ *and a state* σ *iff* `val` *weakly conforms to* `T` *in* Γ *and* σ *and*

- *if* $\mathtt{val}{=}\alpha_j$ *and* $\sigma(\alpha_j) = <<\ \mathtt{v}_1\ \mathtt{C}_1 : \mathtt{val}_1, \ldots \mathtt{v}_n\ \mathtt{C}_n : \mathtt{val}_n\ >>^{\mathtt{C}}$, *then for all labels* $\mathtt{v}$, *classes* $\mathtt{C}'$, *types* $\mathtt{T}'$ *with* $(\mathtt{C}', \mathtt{T}') \in FDecs(\Gamma, \mathtt{C}, \mathtt{v})$, *there exists a* $\mathtt{k} \in \{1, \ldots \mathtt{n}\}$ *such that* $\mathtt{v}_k = \mathtt{v}$, $\mathtt{C}_k = \mathtt{C}'$, *and* $\mathtt{val}_k$ *weakly conforms to* $\mathtt{T}'$ *in* Γ *and* σ;
- *if* $\mathtt{val}{=}\alpha_j$ *and* $\sigma(\alpha_j) = <<\mathtt{val}_0, \ldots \mathtt{val}_{n-1}>>^{\mathtt{T}'[]_1 \ldots []_k}$, *then* $\forall \mathtt{i} \in \{0, \ldots \mathtt{n}-1\}$: $\mathtt{val}_i$ *weakly conforms to* $\mathtt{T}'[]_2 \ldots []_k$.

Furthermore, a state σ conforms to an environment Γ *iff for all identifiers* $\mathtt{x}$, *and integers* $\mathtt{i}$

- *if* $\Gamma(\mathtt{x}) \neq \mathtt{Undef}$ *then* $\sigma(\mathtt{x})$ *conforms to* $\Gamma(\mathtt{x})$ *in* Γ *and* σ;
- *if* $\sigma(\alpha_i) = <<\ \ldots >>^{\mathtt{C}}$, *then* α_i *conforms to* $\mathtt{C}$ *in* Γ *and* σ;
- *if* $\sigma(\alpha_i) = <<\ldots>>^{\mathtt{T}[]_1 \ldots []_n}$, *then* α_i *conforms to* $\mathtt{T}[]_1 \ldots []_n$ *in* Γ *and* σ.

Also, an environment Γ conforms to environment Γ' *iff*

- *for any identifier* $\mathtt{x}$, *if* $\Gamma'(\mathtt{x}) \neq \mathtt{Undef}$, *then* $\Gamma(\mathtt{x}) = \Gamma'(\mathtt{x})$;
- *for any identifier* $\mathtt{x}$, *if* $\Gamma'(\mathtt{x}) = \mathtt{Undef} \neq \Gamma(\mathtt{x})$, *then* $\mathtt{x}$ *is declared in* Γ *as a variable.*

Lemma 9. *Given two environments* Γ, Γ', *where* Γ *conforms to* Γ',

- $\Gamma \vdash \Gamma\ \Diamond \implies \Gamma' \vdash \Gamma'\ \Diamond$;
- *for any program* $\mathtt{p}$: $\Gamma' \vdash \mathtt{p} : \Gamma' \implies \Gamma \vdash \mathtt{p} : \Gamma$;
- *for any term* $\mathtt{t}$, *and type* $\mathtt{T}$: $\Gamma' \vdash \mathtt{t} : \mathtt{T} \implies \Gamma \vdash \mathtt{t} : \mathtt{T}$;
- $\Gamma \vdash \mathtt{T} <_{wdn} \mathtt{T}' \iff \Gamma' \vdash \mathtt{T} <_{wdn} \mathtt{T}'$;
- *for* $\mathtt{T}_1 \ldots \mathtt{T}_n$: $FirstFit(\Gamma, \mathtt{m}, \mathtt{T}_1, \mathtt{T}_2 \times \ldots \times \mathtt{T}_n) = FirstFit(\Gamma', \mathtt{m}, \mathtt{T}_1, \mathtt{T}_2 \times \ldots \times \mathtt{T}_n)$.

Definition 10 *For object* $\mathtt{obj} = <<\mathtt{l}_1\ \mathtt{C}_1 : \mathtt{val}_1, \mathtt{l}_2\ \mathtt{C}_2 : \mathtt{val}_2, \ldots \mathtt{l}_n\ \mathtt{C}_n : \mathtt{val}_n>>^{\mathtt{C}'}$, *state* σ, *value* $\mathtt{val}$, *identifier or reference* $\mathtt{z}$, *class* $\mathtt{C}$, *field identifier* $\mathtt{f}$, *an* $\mathtt{m} \geq 0$, *array* $\mathtt{arr} = <<\mathtt{val}_0, \ldots \mathtt{val}_{n-1}>>^{\mathtt{T}[]_1 \ldots []_m}$ *and integer value* $\mathtt{k}$ *we define:*

- *the* access to field $\mathtt{f}$ declared in class $\mathtt{C}$ *as* $\mathtt{obj(f,C)}$:
  $\mathtt{obj}(\mathtt{f}, \mathtt{C}) = \mathtt{val}_i$ *where* $\mathtt{f} = \mathtt{l}_i$ *and* $\mathtt{C} = \mathtt{C}_i$
- *the* access to component $\mathtt{f}, \mathtt{C}$ of an object *stored at a reference* $\mathtt{z}$ *in state* σ :
  $\sigma(\mathtt{z}, \mathtt{f}, \mathtt{C}) = \sigma(z)(f, C)$
- *the* access to the k^{th} component *of* $\mathtt{arr}$, $\mathtt{arr[k]}$:
  $\mathtt{arr[k]} = \mathtt{val}_k$ *if* $0 \leq \mathtt{k} \leq \mathtt{n}-1$
  $\mathtt{arr[k]} = \mathsf{IndOutBndExc}$ *otherwise*
- *a new state,* $\sigma' = \sigma[\mathtt{z} \mapsto \mathtt{val}]$, *such that*:
  $\sigma'(\mathtt{z}) = \mathtt{val}$
  $\sigma'(\mathtt{z}') = \sigma(\mathtt{z}')$ *for* $\mathtt{z}' \neq \mathtt{z}$:
- *a new object,* $\mathtt{obj}' = \mathtt{obj}[\mathtt{f}, \mathtt{C} \mapsto \mathtt{val}]$, *and a new state,* $\sigma' = \sigma[\mathtt{z}, \mathtt{f}, \mathtt{C} \mapsto \mathtt{val}]$:
  $\mathtt{obj}'(\mathtt{f}, \mathtt{C}) = \mathtt{val}$
  $\mathtt{obj}'(\mathtt{f}', \mathtt{C}') = \mathtt{obj}(\mathtt{f}', \mathtt{C}')$ *if* $\mathtt{f} \neq \mathtt{f}'$ *or* $\mathtt{C} \neq \mathtt{C}'$
  $\sigma' = \sigma[\mathtt{z} \mapsto \sigma(\mathtt{z})[\mathtt{f}, \mathtt{C} \mapsto \mathtt{val}]]$
- *a new array,* $\mathtt{arr}' = \mathtt{arr}[\mathtt{k} \mapsto \mathtt{val}]$, *and a new state,* $\sigma' = \sigma[\mathtt{arr}, \mathtt{k} \mapsto \mathtt{val}]$:

$$
\begin{array}{lll}
\mathtt{arr}'[\mathtt{k}] & = & \mathtt{val} \\
\mathtt{arr}'[\mathtt{j}] & = & \mathtt{arr}[\mathtt{j}] \qquad\qquad \textit{if } \mathtt{j} \neq \mathtt{k} \\
\sigma' & = & \sigma[\mathtt{arr} \mapsto \mathtt{arr}[\mathtt{k} \mapsto \mathtt{val}]]
\end{array}
$$

$$
\frac{< \mathtt{stmts}, \sigma > \leadsto_{\mathtt{p}} < \sigma' >}{< \mathtt{stmts}; \mathtt{stmt}, \sigma > \leadsto_{\mathtt{p}} < \mathtt{stmt}, \sigma' >}
\qquad
\frac{< \mathtt{stmts}, \sigma > \leadsto_{\mathtt{p}} < \mathtt{stmts}', \sigma' >}{< \mathtt{stmts}; \mathtt{stmt}, \sigma > \leadsto_{\mathtt{p}} < \mathtt{stmts}'; \mathtt{stmt}, \sigma' >}
$$

$$
\frac{< \mathtt{e}, \sigma > \leadsto_{\mathtt{p}} < \mathtt{e}', \sigma' >}{< \mathtt{if\ e\ then\ stmts\ else\ stmts}', \sigma > \leadsto_{\mathtt{p}} < \mathtt{if\ e}'\ \mathtt{then\ stmts\ else\ stmts}', \sigma' >}
\qquad
\frac{}{< \mathtt{if\ true\ then\ stmts\ else\ stmts}', \sigma > \leadsto_{\mathtt{p}} < \mathtt{stmts}, \sigma >}
$$

$$
\frac{}{< \mathtt{if\ false\ then\ stmts\ else\ stmts}', \sigma > \leadsto_{\mathtt{p}} < \mathtt{stmts}', \sigma >}
\qquad
\frac{}{< \mathtt{return}, \sigma > \leadsto_{\mathtt{p}} < \sigma >}
$$

$$
\frac{< \mathtt{e}, \sigma > \leadsto_{\mathtt{p}} < \mathtt{e}', \sigma' >}{< \mathtt{return\ e}, \sigma > \leadsto_{\mathtt{p}} < \mathtt{return\ e}', \sigma' >}
\qquad
\frac{\mathtt{val}\ \text{is ground}}{< \mathtt{return\ val}, \sigma > \leadsto_{\mathtt{p}} < \mathtt{val}, \sigma >}
$$

Fig. 17. statements

We distinguish ground terms which cannot be further rewritten, and l-ground terms, which are "almost ground", since they may not be further rewritten if they appear on the left hand side of an assignment:

Definition 11 *A Java$_{se}$ term* `t` *is*

- ground *iff* `t` *is a primitive value, or if* `t`=$\alpha_{\mathtt{i}}$ *for some* `i`;
- l-ground *iff* `t` *is ground, or* `t=id` *for some identifier* `id`, *or* `t=` $\alpha_{\mathtt{i}}$`.[C]f` *for a class* `C` *and a field* `f` *and integer value* `i`, *or* `t` $= \alpha_{\mathtt{i}}$`[k]` *for some integer values* `i` *and* `k`.

4.2 Program execution

Figures 17-20 describe the operational semantics of Java$_{se}$, in terms of the $\leadsto_{\mathtt{p}}$ -relationship.

Figure 17 describes the execution of statements. Statement sequences are evaluated from left to right. In conditional statements the condition is evaluated first; if it evaluates to `true`, then the first branch is executed, otherwise the second. A `return` statement terminates execution. A statement returning an expression evaluates this expression until ground and replaces itself by this ground value – thus modelling methods returning values.

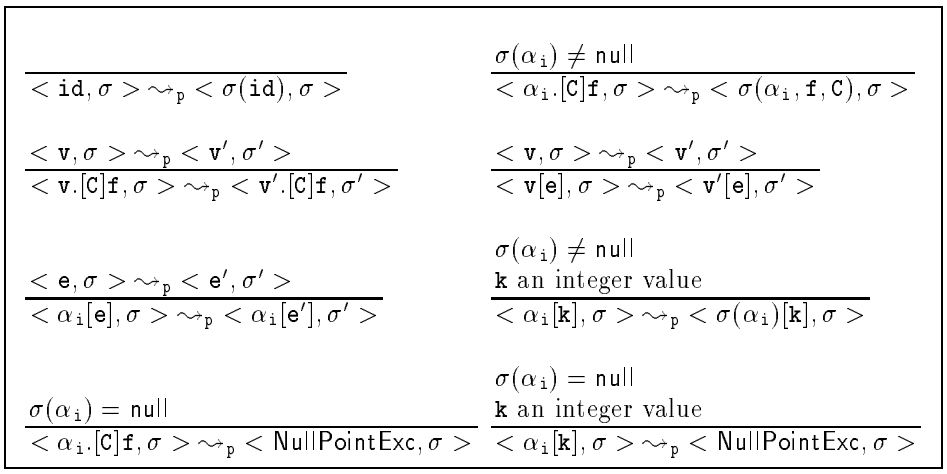

Fig. 18. variables

In figure 18 we describe the evaluation of variables, field access and array access. Variables (*i.e.* identifiers, instance variable access or array access) are evaluated from left to right. The rules about assignment in 20 will prevent an expression like $\mathtt{x}$ or $\alpha_i.[\mathtt{C}]\mathtt{v}$ from being rewritten any further if it is the left hand side of an assignment. They would allow an expression of the form `u[C1].w[C2].x[C3].y` to be rewritten to an expression of the form α_j`[C3].y` for some `j`. Furthermore, there is *no* rule of the form $< \alpha_j, \sigma > \rightsquigarrow_p < \sigma(\alpha_j), \sigma >$. This is because there is no explicit dereferencing operator in Java. Objects are passed as references, and they are dereferenced only implicitly, when their fields are accessed.
Array access as described here adheres to the rules in ch. 15.12 of [13], which require full evaluation of the expression to the left of the brackets. Thus, with our operational semantics, `a[(a = b)[3]]` corresponds to `a[b[3]]; a = b`.
In figure 19 we describe the creation of new objects or arrays, cf. ch. 15.8-15.9 of [13]. Essentially, a new value of the appropriate array or class type is created, and its address is returned. The fields of the array, and the components the object are assigned initial values (as defined in ch. 4.5.5. of [13]) of the type to which *they* belong.

Definition 12 *The* initial value *of a simple type is defined as follows*:

- `0` *is the initial value of* `int`
- `' '` *is the initial value of* `char`
- `false` *is the initial value of* `bool`
- `null` *is the initial value of any class or interface*

Figure 20 describes the evaluation of assignments. The left hand side is evaluated first, until it becomes l-ground. Then the right hand side is evaluated, up to the

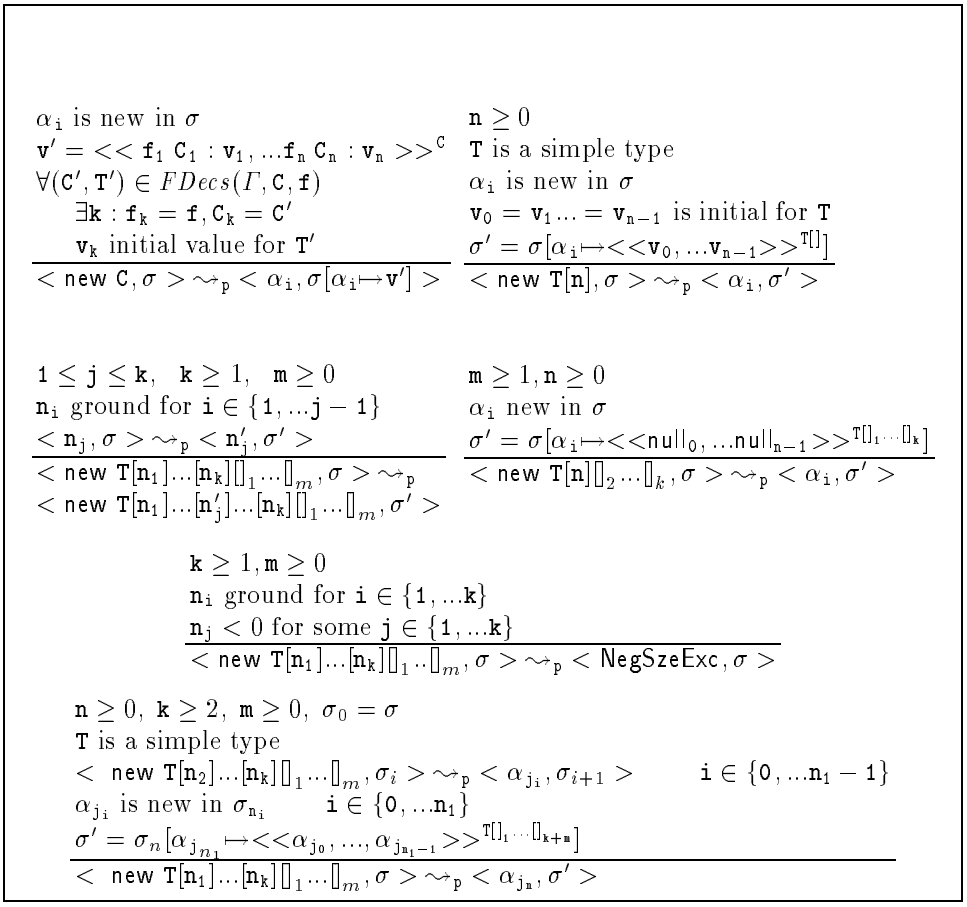

Fig. 19. object and array creation

point of obtaining a ground term. Then the state is modified accordingly. Note that we have *no* rule of the form $< \alpha_{\mathtt{j}} := \mathtt{value}, \sigma > \rightsquigarrow_{\mathtt{p}} \ldots$. This is because in Java overwriting of objects is not possible – only sending messages to them, or overwriting selected instance variables.

Figure 21 describes the evaluation of method calls. Expressions are evaluated left to right, cf ch. 9.3 in [17]. The first rule describes rewriting the $\mathtt{k}^{th}$ expression, where all the previous expressions (*i.e.* $\mathtt{e_i}, \mathtt{i} \in \{\mathtt{1}, \ldots \mathtt{k}-\mathtt{1}\}$) are ground. The second rule describes dynamic method look up, taking into account the argument types, and the statically calculated method descriptor $\mathtt{AT}$, and where $\mathtt{t}[\mathtt{t}'/\mathtt{x}]$ has the usual meaning of replacing the variable $\mathtt{x}$ by the term $\mathtt{t}'$ in the term $\mathtt{t}$.

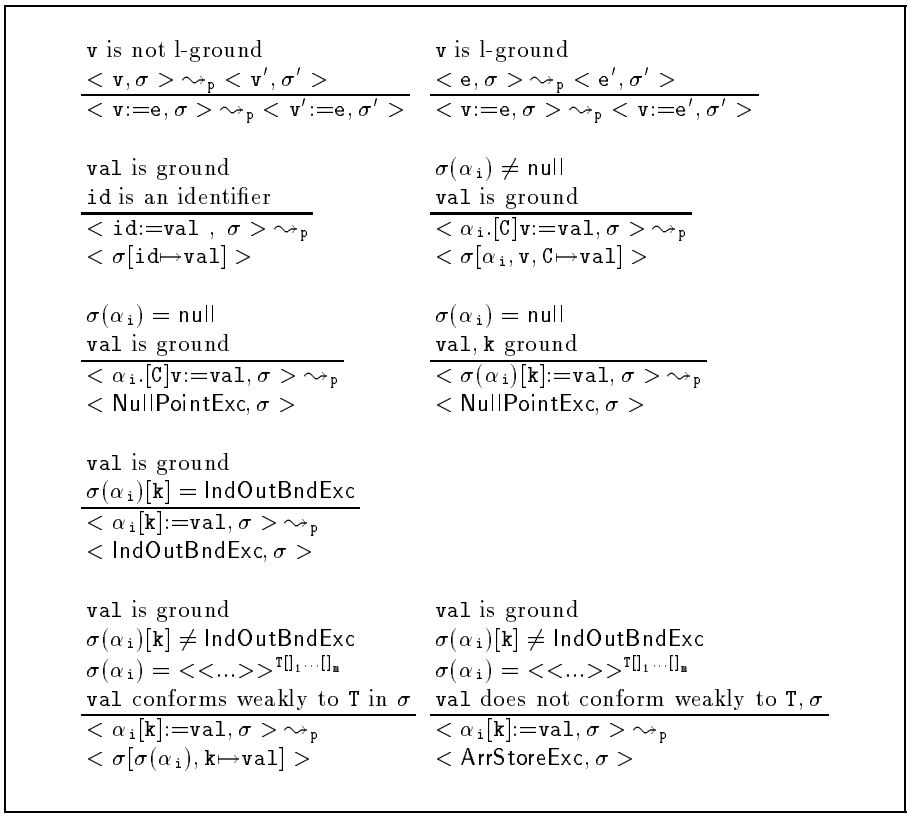

Fig. 20. assignment

4.3 Properties of the operational semantics

The operational semantics is deterministic:

Lemma 10. *For any configuration with a state that conforms to the environment, and any Java$_{se}$ term, the relation $\leadsto_{\mathrm{p}}$ determines at most one step.*

Lemma 11. *For any Java$_{se}$ term* $\mathtt{t}$*, state* σ *and environment* Γ*, if* $\mathtt{t}$ *does not contain an assignment to* α_{i} *as a subterm, and* $< Comp\{[\mathtt{t}, \Gamma]\}, \sigma > \leadsto_{\mathrm{p}} < \mathtt{t}', \sigma >$*, then* $\mathtt{t}'$ *does not contain an assignment to* α_{i} *as a subterm.*

5 Soundness of the Java$_s$ type system

The subject reduction theorem says, that any well-typed Java$_{se}$ term either rewrites to a term which will lead to an exception or rewrites to another well-typed term of a type that can be widened to the type of the original term.

$$\frac{\mathtt{e}_i \text{ is ground, for all } \mathtt{i} \in \{1,\ldots \mathtt{k}-1\}, \mathtt{n} \geq \mathtt{k} \geq 1 \quad < \mathtt{e}_k, \sigma > \rightsquigarrow_p < \mathtt{e}'_k, \sigma' >}{< \mathtt{e}_1.[\mathtt{AT}]\mathtt{m}(\mathtt{e}_2, \ldots, \mathtt{e}_k, \ldots \mathtt{e}_n), \sigma > \rightsquigarrow_p < \mathtt{e}_1.[\mathtt{AT}]\mathtt{m}(\mathtt{e}_2, \ldots, \mathtt{e}'_k, \ldots \mathtt{e}_n), \sigma' >}$$

$$\frac{\begin{array}{l}\mathtt{val}_i \text{ is ground } \mathtt{i} \in \{1,\ldots \mathtt{n}\}, \mathtt{n} \geq 1 \\ \sigma(\mathtt{val}_1) = <<\ldots>>^{\mathtt{C}} \\ \mathtt{AT} = \mathtt{T}_2 \times \ldots \times \mathtt{T}_n \\ \mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{C}, \mathtt{p}) = (\mathtt{C}', \lambda \mathtt{x}_2 : \mathtt{T}_2 \ldots \lambda \mathtt{x}_n : \mathtt{T}_n.\{\ \mathtt{stmts}\ \}) \\ \mathtt{z}_i \text{ are new identifiers in } \sigma \\ \sigma' = \sigma[\mathtt{z}_1 \mapsto \mathtt{val}_1] \ldots [\mathtt{z}_n \mapsto \mathtt{val}_n] \\ \mathtt{stmts}' = \mathtt{stmts}[\mathtt{z}_1/\mathtt{this}, \mathtt{z}_2/\mathtt{x}_2, \ldots \mathtt{z}_n/\mathtt{x}_n]\end{array}}{< \mathtt{val}_1.[\mathtt{AT}]\mathtt{m}(\mathtt{val}_2, \ldots \mathtt{val}_n), \sigma > \rightsquigarrow_p < \mathtt{stmts}', \sigma' >}$$

Fig. 21. method calls

Theorem 1 Subject Reduction *For a state σ that conforms to an environment Γ, a Java$_{se}$ program* $\mathtt{p}$ *with* $\Gamma \vdash \mathtt{p} : \Gamma$, *a non-ground Java$_{se}$ term* $\mathtt{t}$ *that contains no assignments of the form* $\alpha_i := \ldots$, *and type* $\mathtt{T}$ *with* $\Gamma, \sigma \vdash \mathtt{t} : \mathtt{T}$, *there exist* σ', $\mathtt{t}'$ *such that:*

- $< \mathtt{t}, \sigma > \rightsquigarrow_p < \mathtt{t}', \sigma' >$, *and*
  - $\exists \mathtt{t}'', \sigma'' : < \mathtt{t}', \sigma' > \rightsquigarrow_p{}^* < \mathtt{t}'', \sigma'' >$ *and* $\mathtt{t}''$ *contains an exception*
    or
  - $\exists \Gamma', \mathtt{T}' :$ Γ' *conforms to* Γ, σ' *conforms to* Γ', *and* $\Gamma', \sigma' \vdash \mathtt{t}' : \mathtt{T}'$, *and* $\Gamma \vdash \mathtt{T}' <_{wdn} \mathtt{T}$

  or
- $< \mathtt{t}, \sigma > \rightsquigarrow_p < \sigma' >$ *and* σ' *conforms to* Γ

Furthermore, if $\mathtt{t}$ *is a variable, and not l-ground, then* $< \mathtt{t}, \sigma > \rightsquigarrow_p < \mathtt{t}', \sigma' >$ *and* $\mathtt{t}'$ *is not ground. Also, if* $\mathtt{t}$ *is not l-ground and not an array access, then* $\mathtt{T} = \mathtt{T}'$.

Theorem 2 Soundness *Take any Java$_s$ term* $\mathtt{t}$, *a well-formed environment* Γ, *a type* $\mathtt{T}$ *with* $\Gamma \vdash \mathtt{t} : \mathtt{T}$ *a Java$_s$ program* $\mathtt{p}$ *with* $\Gamma \vdash \mathtt{p} : \Gamma$, *and a state* σ *that conforms to* Γ. *Then there exists a Java$_{se}$ program* $\mathtt{p}'$, $\mathtt{p}' = \mathit{Comp}\{[\mathtt{p}, \Gamma]\}$, *a Java$_{se}$ term term* $\mathtt{t}'$, *and a state* σ', *such that:*

- *The execution of* $< \mathit{Comp}\{[\mathtt{t}, \Gamma]\}, \sigma > \rightsquigarrow_{\mathtt{p}'}{}^*$ *does not terminate*
  or
- $< \mathit{Comp}\{[\mathtt{t}, \Gamma]\}, \sigma > \rightsquigarrow_{\mathtt{p}'}{}^* < \mathtt{t}', \sigma' >$ *and* $\mathtt{t}'$ *contains an exception as a subterm*
  or
- $\mathtt{T} \neq \mathtt{void}$, *and* $< \mathit{Comp}\{[\mathtt{t}, \Gamma]\}, \sigma > \rightsquigarrow_{\mathtt{p}'}{}^* < \mathtt{t}', \sigma' >$ *and* $\mathtt{t}'$ *is ground, and* $\exists \mathtt{T}' :$ $\Gamma, \sigma' \vdash \mathtt{t}' : \mathtt{T}'$, $\Gamma \vdash \mathtt{T}' <_{wdn} \mathtt{T}$ *and* σ' *conforms to* Γ

or

– $\mathtt{T} = \mathtt{void}$, *and* $< Comp\{[\mathtt{t}, \Gamma]\}, \sigma > \rightsquigarrow_{\mathrm{p}'}{}^{*} < \sigma' >$ *and* σ' *conforms to* Γ

6 An example

The following, admittedly contrived, Java program serves to demonstrate the concepts introduced in the previous sections. It can have the following interpretation: Philosophers like philosophers. When a philosopher thinks about a problem together with another philosopher, then, after some deliberation they refer the problem to a third philosopher. When a philosopher thinks together with a French philosopher, they produce a book. French philosophers like food, and when they think together with another philosopher, they finally refer the question to a French philosopher (this way of method overriding is allowed in [17]).

```
class Phil {
        Phil like ;
        Phil think(Phil y){ ... }
        Book think(FrPhil y){ ... }
 }
class FrPhil extends Phil {
        Food like ;
        FrPhil think(Phil y){ like = oyster ; ... }
FrPhil aPhil ; FrPhil pascal ;
... pascal.like ; pascal.think(pascal) ; pascal.think(aPhil) ;
... aPhil.like ; aPhil.think(pascal) ; aPhil.think(aPhil) ;
aPhil = pascal ;
... aPhil.like ; aPhil.think(pascal) ; aPhil.think(aPhil) ;
```

The functions $FDec(,,)$, $FDecs(,,)$, $MSigs(,,)$, and $MDecs(,,)$ are as follows:

$FDec(\Gamma_0, \mathtt{Phil}, \mathtt{like})$ = (Phil, Phil)
$FDec(\Gamma_0, \mathtt{FrPhil}, \mathtt{like})$ = (FrPhil, Food)

$FDecs(\Gamma_0, \mathtt{Phil}, \mathtt{like})$ = { (Phil, Phil) }
$FDecs(\Gamma_0, \mathtt{FrPhil}, \mathtt{like})$ = { (Phil, Phil),
(FrPhil, Food) }

$MDecs(\Gamma_0, \mathtt{Phil}, \mathtt{think})$ = { (Phil, Phil→ Phil),
(Phil, FrPhil → Book) }
$MDecs(\Gamma_0, \mathtt{FrPhil}, \mathtt{think})$ = { (FrPhil, Phil→ FrPhil),
(Phil, FrPhil→ Book) }

$MSigs(\Gamma_0, \mathtt{Phil}, \mathtt{think})$ = { Phil→ Phil, FrPhil→ Book }
$MSigs(\Gamma_0, \mathtt{FrPhil}, \mathtt{think})$ = { Phil→ FrPhil, FrPhil→ Book }

The corresponding Java$_s$ environment Γ_0 is:

Γ_0 = `Phil ext Object { like : Phil, think : Phil → Phil,`
`think : FrPhil → Book, }`
`FrPhil ext Phil { like: Food, think : Phil → FrPhil }`

The corresponding Java$_s$ program is `p`:

```
p = {Phil ext Object { think is λ y:Phil.{ ... },
          think is λ y:Phil.{ ... } },
     FrPhil ext Phil { think is
          λ y:FrPhil.{ this.like := oyster ; ... } }
... }
```

The program `p` would be mapped to `p'`, the following Java$_{se}$ program:

```
p' = Comp{[p, Γ0]} = {
     Phil ext Object { think is λ y:Phil. { ... },
          think is λ y:FrPhil. { ... } }
     FrPhil ext Phil { think is λ y:FrPhil.
          { this.[FrPhil]like := oyster ; ... } }
 pascal.[Phil]like ; pascal.[FrPhil]think(pascal) ;
          pascal.[Phil]think(aPhil) ;
 aPhil.[Phil]like ; aPhil.[FrPhil]think(pascal) ;
          aPhil.[Phil]think(aPhil) ;
aPhil = pascal ;
 aPhil.[Phil]like ; aPhil.[FrPhil]think(pascal) ;
          aPhil.[Phil]think(aPhil) ;
 }
```

The state σ_0 conforms to the environment Γ_0:
σ_0(`aPhil`) = α_2
$\sigma_0(\alpha_2)$ = `<< like Phil: `α_4`, like FrPhil: croissant >>`$^{\texttt{FrPhil}}$
$\sigma_0(\alpha_4)$ = `<< x Phil: null >>`$^{\texttt{Phil}}$

Execution of the method call `aPhil.[Phil]think(aPhil)` results in the following rewrites
$< \texttt{aPhil.[Phil]think(aPhil)}, \sigma_0 > \rightsquigarrow_{\texttt{p}'} < \alpha_2\texttt{.[Phil]think(aPhil)}, \sigma_0 > \rightsquigarrow_{\texttt{p}'}$
$< \alpha_2\texttt{.[Phil]think}(\alpha_2), \sigma_0 > \rightsquigarrow_{\texttt{p}'} < (\texttt{w.[FrPhil]like:=oyster}; \ldots), \sigma_1 > \rightsquigarrow_{\texttt{p}'}$
$< (\ldots), \sigma_2 >$
where σ_1, σ_2 are:

$\sigma_1(\texttt{aPhil}) = \sigma_0(\texttt{aPhil}) = \alpha_2$
$\sigma_1(\texttt{w}) = = \alpha_2$
$\sigma_1(\texttt{w}') = = \alpha_2$
$\sigma_1(\alpha_2) = \sigma_0(\alpha_2) = \texttt{<<like Phil:}\alpha_4\texttt{, like FrPhil:croissant>>}^{\texttt{FrPhil}}$
$\sigma_1(\alpha_4) = \sigma_1(\alpha_4) = \texttt{<<like Phil:null >>}^{\texttt{Phil}}$
$\sigma_2(\texttt{aPhil}) = \sigma_1(\texttt{aPhil}) = \alpha_2$
$\sigma_2(\texttt{w}) = \sigma_1(\texttt{w}) = \alpha_2$
$\sigma_2(\texttt{w}') = \sigma_1(\texttt{w}') = \alpha_2$
$\sigma_2(\alpha_2) = = \texttt{<< like Phil:}\alpha_4\texttt{, like FrPhil:oyster >>}^{\texttt{FrPhil}}$
$\sigma_2(\alpha_4) = \sigma_1(\alpha_4) = \texttt{<< like Phil: null >>}^{\texttt{Phil}}$

If we consider the "environment extension" as in theorem 1, then in the third step of the reductions, we would have the environment $\Gamma' = \Gamma_0, \texttt{w} : \texttt{FrPhil}, \texttt{w}' : \texttt{FrPhil}$. The states σ_1, σ_2 conform to Γ'.

Execution of an array creation expression `new int[1][2][][]`, for the state σ_0: $<$ `new int[1][2][][]`$, \sigma_0 > \rightsquigarrow_{\texttt{p}'} < \alpha_7, \sigma_5 >$ where α_5 and α_6 are new in σ_0, and have the following contents in σ_5:

$\sigma_5(\alpha_5) = \texttt{<<null, null, null>>}^{\texttt{int[][][]}}$
$\sigma_5(\alpha_6) = \texttt{<<null, null, null>>}^{\texttt{int[][][]}}$
$\sigma_5(\alpha_7) = \texttt{<<}\alpha_5, \alpha_6\texttt{>>}^{\texttt{int[][][][]}}$

7 Conclusions and future work

We have given a formal description of the operational semantics and type system for a substantial subset of Java. We consider this subset to contain many of the features which together might have led to difficulties in the Java type system. By applying some simplifications we obtained a straightforward system which we believe does not diminish the application of our results.
We aim to extend the language subset to describe a larger part of Java, and we also hope that our approach may serve as the basis for other studies on the language and possible extensions.

8 Acknowledgments

We are greatly indebted to several people, who read previous versions of this work and gave us valuable feedback and uncovered flaws: Peter Sellinger, David von Oheimb, Safraz Kurshid, Donald Syme, Yao Feng, Steve Vickers, an anonymous FOOL4 referee for feedback, and Guiseppe Castagna.
We would also like to express our appreciation to Bernie Cohen for awakening our interest in the application of formal methods to Java and especially to all our students whose overwhelming interest in Java convinced us that this work needed to be undertaken.

References

1. M. Abadi and L. Cardelli. A semantics of object types. In *LICS'94 Proceedings*, 1994.
2. Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized types and Java. In *POPL'97 Proceedings*, January 1997.
3. Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice & Experience*, 25(8):863–889, August 1995.
4. John Boyland and Giuseppe Castagna. Type-safe compilation of covariant specialization: A practical case. In *ECOOP'96 Proceedings*, July 1996.
5. P. Canning, William Cook, and William Olthoff. Interfaces for object-oriented programming. In *OOPLSA'89*, pages 457–467, 1989.
6. Giuseppe Castagna. Parasitic Methods: Implementation of Multimethods for Java. Technical report, C.N.R.S, November 1996.
7. Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 15 February 1995.
8. William Cook. A Proposal for making Eiffel Type-safe. In S. Cook, editor, *ECOOP'87 Proceedings*, pages 57–70. Cambridge University Press, July 1989.
9. William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *POPL'90 Proceedings*, January 1990.
10. Luis Damas and Robin Milner. Principal Type Schemes for Functional Languages. In *POPL'82 Proceedings*, 1982.
11. Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.
12. Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? In *Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997.
13. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996.
14. R. Harper. A simplified account of polymorphic references. Technical Report CMU-CS-93-169, Carnegie Mellon University, 1993.
15. Daniel Ingalls. The smalltalk-76 programming system design and implementation. In *POPL'78 Proceedings*, pages 9–15, January 1978.
16. The Java language specification, October 1995.
17. The Java language specification, May 1996.
18. Bertrand Meyer. Static typing and other mysteries of life, December 1995.
19. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *POPL'97 Proceedings*, January 1997.
20. Peter Sellinger. private communication, October 1996.
21. Mads Tofte. Type Inference for Polymorphic References. In *Information and Computation'80 Conference Proceedings*, pages 1–34, November 1980.