

Is the Java Type System Sound?

Sophia Drossopoulou and Susan Eisenbach
Department of Computing
Imperial College
email: sd and se @doc.ic.ac.uk

Abstract

We argue that the Java type system is sound, by proving a subject reduction theorem. We define a subset of Java, a language which is safe and which reflects the most essential features of Java, a term rewriting system for the 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.

1 Introduction

1.1 Motivation

Java [17] is a rapidly spreading programming language which aims to support safe distributed programming on the Internet. According to its developers[17], it is a

“simple robust object-oriented platform independent multi-threaded dynamic general purpose programming environment. It’s best for creating applets and applications for the Internet, intranets and any other complex, distributed network.”

Even before the first complete language description was available [14] use of the language was extremely widespread and the rate of increase in usage is steep. It looks like the language may not have reached a stable point in its development yet: there exist differences between the language descriptions [18, 19, 14], and there are many suggestions for additional features [22, 3]. Several studies have uncovered flaws in the security of the Java system [12], 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 [10, 6] is a simplification of the signatures extension for C++ [4] 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 (Algol68 [25], Pascal [2], Modula-2 [21], C++ partly [13]). 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 [9, 20]. 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 in Java is sound, in the sense that the evaluation of any expression will produce a value of a type “compatible” with the type assigned to it by the type system.

1.2 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, and dynamic method binding.

We chose the Java subset from above because we consider the Java way of combining classes, interfaces and dynamic method binding to be both novel and interesting. Furthermore, we started with an imperative subset right from the start, because the extension of type systems to the imperative case has sometimes uncovered new problems, (*e.g.* multimethods for functional languages [8], and for imperative languages in [5], the Damas and Milner polymorphic type systems for functional languages [11], and for the imperative extension [24]). We plan to extend our study, starting with arrays and the associated dynamic checks, until we either have covered all of Java – or we have uncovered loopholes in the type system.

We aim to describe the language as in the [14] definition with the exception of method binding, which we model as described in [19], because it imposes a weaker requirement. Namely, [19] 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 [14] 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 [14].

1.3 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.* 3.5. 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 [14]). 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, *c.f.* section 3.11.1. 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 (*i.e.* no local variables, no blocks) also allow the definition of the the state as a flat structure, where global variables are mapped to primitive values or addresses and addresses are mapped to objects. Objects carry their classes (similar to the Smalltalk abstract machine [16], thus we do not need store types [1], or location typings [15]). Objects are labelled tuples, where each label contains the class in which it was declared.

As an interesting subsidiary result, we formulated two important properties which any compile-time correct Java program must satisfy

- the requirement for any class, which is widening another interface or class, to provide an implementation for any method declared in that superinterface or superclass
- the substitution property for enriched expressions

We believe that these properties were in the language designers' minds, although not explicitly stated in [14].

This paper is organised 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 and the `null` value. 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 types, instance and class variable types, parameter and result types for methods, and interfaces of classes. The evaluation information is the 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.

```

Program ::= { ( ClassBody )* }
ClassBody ::= ClassId ext ClassName { ( MethodBody )* }

```

2.1.1 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 `Javas` operational semantics without restricting the expressiveness, since it requires only a minor transformation of any Java method body to satisfy this property.

```

MethodBody ::= MethId is (λ ParId: VarType.)* { Stmts ; return [Expr] }

```

2.1.2 Statements

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; **try** and **throw** statements belong to exceptions which are outside the scope of our current investigations.

```

Stmts ::= ε | Stmts ; Stmt
Stmt ::= if Expr then Stmts else Stmts
      | Var := Expr
      | Expr

```

2.1.3 Expressions, variables and values

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. Pointers to objects are implicit.¹

```

Expr ::= Value
      | Var
      | Expr.MethName ( Expr* )
Var ::= Name
      | Var.VarName
      | this
Value ::= PrimValue | RefValue
PrimValue ::= intValue | charValue | byteValue | ...
RefValue ::= null

```

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 [14], 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 so they are found in the program part.

¹They are only necessary for the operational semantics, and will be introduced in section 4, as part of the enriched language `Javasee`.

An interface declaration introduces a new interface as a sequence of components. Interfaces may be extensions of other interfaces. The only interface components in Java_s are methods, because interface variables are implicitly static, and we have not yet considered static variables. Variable declarations introduce new variables of a given type.

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

2.2.1 The subclass \sqsubseteq and implements $:_{impl}$ relationships

The following inference rules define the subclass relationship \sqsubseteq .

- Every class introduced in Γ is its own subclass. An assertion such as $\Gamma \vdash C \sqsubseteq C$ indicates that C is defined in the environment Γ as a class:

$$\frac{}{\Gamma, C \text{ ext } C' \text{ impl } \dots\{\dots\}, \Gamma' \vdash C \sqsubseteq C}$$

- The direct superclass of a class is indicated in its declaration:

$$\frac{}{\Gamma, C \text{ ext } C' \text{ impl } \dots\{\dots\}, \Gamma' \vdash C \sqsubseteq C'}$$

- The assertion $\Gamma \vdash C :_{impl} I$ indicates that the class C was declared in Γ as providing an implementation for interface I :

$$\frac{}{\Gamma, C \text{ ext } C' \text{ impl } \dots I \dots\{\dots\}, \Gamma' \vdash C :_{impl} I}$$

- `Object` is a predefined class:

$$\frac{}{\vdash Object \sqsubseteq Object}$$

- The subclass relationship is transitive:

$$\frac{\Gamma \vdash C \sqsubseteq C' \quad \Gamma \vdash C' \sqsubseteq C''}{\Gamma \vdash C \sqsubseteq C''}$$

2.2.2 The subinterface relationship \leq

The following inference rules define the subinterface \leq relationship.

- Every interface is its own subinterface and the assertion $\Gamma \vdash I \leq I$ also indicates that I is defined in the environment Γ as an interface:

$$\frac{}{\Gamma, I \text{ ext } \dots\{\dots\}, \Gamma' \vdash I \leq I}$$

- The superinterface of an interface is indicated in its declaration:

$$\frac{}{\Gamma, \mathbf{I} \text{ ext } \dots, \mathbf{I}', \dots \{ \dots \}, \Gamma' \vdash \mathbf{I} \leq \mathbf{I}'}$$

- The subinterface relationship is transitive:

$$\frac{\frac{\Gamma \vdash \mathbf{I} \leq \mathbf{I}' \quad \Gamma \vdash \mathbf{I}' \leq \mathbf{I}''}{\Gamma \vdash \mathbf{I} \leq \mathbf{I}''}}$$

2.2.3 The widening relationship $<_{wdn}$

The widening relationship exists between variable types. If a type \mathbf{T} can be widened to a type \mathbf{T}' (expressed as $\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{T}'$), then a value of type \mathbf{T} can be assigned to a variable of type \mathbf{T}' without any run-time casting or checking taking place. This is defined in chapter 5.1.4 [19]; chapter 5.1.2 in [19] 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 `nullT` which can be widened to any other type.

$$\frac{\Gamma \vdash \mathbf{T} \diamond_{VarType}}{\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{T}} \quad \frac{\Gamma \vdash \mathbf{T} \sqsubseteq \mathbf{T}'}{\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{T}'}$$

$$\frac{\Gamma \vdash \mathbf{T} \leq \mathbf{T}'}{\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{T}'} \quad \frac{\Gamma \vdash \mathbf{T} \sqsubseteq \mathbf{T}' \quad \Gamma \vdash \mathbf{T}' :_{impl} \mathbf{T}'' \quad \Gamma \vdash \mathbf{T}'' \leq \mathbf{T}'''}{\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{T}'''}$$

$$\frac{\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{T}}{\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{Object}} \quad \frac{}{\Gamma \vdash \mathbf{nullT} <_{wdn} \mathbf{nullT}}$$

$$\frac{\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{T}}{\Gamma \vdash \mathbf{nullT} <_{wdn} \mathbf{T}}$$

2.3 Types

We distinguish variable types (sets of possible run-time values for variables) and method types:

$$\begin{aligned} VarType & ::= PrimType \mid ClassName \mid InterfaceName \\ PrimType & ::= \mathbf{bool} \mid \mathbf{char} \mid \mathbf{int} \mid \dots \\ MethType & ::= ArgType \rightarrow (VarType \mid \mathbf{void}) \\ ArgType & ::= (VarType (\times VarType)^*) \end{aligned}$$

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

- $Args(\mathbf{MT}) = \mathbf{T}_1 \times \dots \times \mathbf{T}_n$
- $Res(\mathbf{MT}) = \mathbf{T}$

Variable types (*i.e.* the primitive types, the interfaces and classes) are required in type declarations, whereas method types (*i.e.* n argument types, and a result type, with $n \geq 0$) are required in method declarations:

$$\frac{\Gamma \vdash \mathbf{C} \sqsubseteq \mathbf{C}}{\Gamma \vdash \mathbf{C} \diamond_{VarType}} \quad \frac{\Gamma \vdash \mathbf{I} \leq \mathbf{I}}{\Gamma \vdash \mathbf{I} \diamond_{VarType}} \quad \frac{}{\vdash \mathbf{int} \diamond_{VarType}} \quad \frac{}{\vdash \mathbf{char} \diamond_{VarType}} \quad \frac{}{\vdash \mathbf{bool} \diamond_{VarType}}$$

$$\begin{array}{c}
\Gamma \vdash T \diamond_{VarType} \\
\Gamma \vdash T_i \diamond_{VarType} \quad i \in \{1, \dots, n\}, n \geq 0 \\
\hline
\Gamma \vdash T_1 \times \dots \times T_n \diamond_{ArgType} \\
\Gamma \vdash T_1 \times \dots \times T_n \rightarrow T \diamond_{MethType}
\end{array}$$

2.4 Well formed declarations and environments

It is easy to see that the relations \sqsubseteq , $:\text{impl}$, \leq and $<_{wdn}$ are computable for any environment. In this section we describe the Java requirements for variable, class or 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 **C1** uses the class **C2** whose declaration follows that of **C1**). 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 , $:\text{impl}$, \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:

$$\overline{\Gamma \vdash \epsilon \diamond}$$

We need the notion of definition table lookup, *i.e.* $\Gamma(\text{Id})$, 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 $\Gamma(\text{id})$ as follows:

- $\Gamma(\mathbf{x}) = T$ iff $\Gamma = \Gamma_1, \mathbf{x} : T, \Gamma_2$
- $\Gamma(\mathbf{C}) = \mathbf{C} \text{ ext } \mathbf{C}' \text{ impl } I_1, \dots, I_n \{v_1 : T_1, \dots, v_m : T_m, m_1 : MT_1, \dots, m_k : T_k\}$ iff $\Gamma = \Gamma_1, \mathbf{C} \text{ ext } \mathbf{C}' \text{ impl } I_1, \dots, I_n \{v_1 : T_1, \dots, v_m : T_m, m_1 : MT_1, \dots, m_k : T_k\}, \Gamma_2$
- $\Gamma(\mathbf{I}) = \mathbf{I} \text{ ext } I_1, \dots, I_n \{m_1 : MT_1, \dots, m_k : MT_k\}$ iff $\Gamma = \Gamma_1, \mathbf{I} \text{ ext } I_1, \dots, I_n \{m_1 : MT_1, \dots, m_k : MT_k\}, \Gamma_2$
- $\Gamma(\mathbf{I}) = \text{Undef}$ otherwise

2.4.1 Variable declarations

A variable should be declared to be of a variable type and there should be declared only once.

$$\begin{array}{c}
\Gamma \vdash \Gamma' \diamond \\
\Gamma \vdash T \diamond_{VarType} \\
\Gamma'(\mathbf{x}) = \text{Undef} \\
\hline
\Gamma \vdash \Gamma', \mathbf{x} : T \diamond
\end{array}$$

Note, that the rule from above allows the type declaration for T to follow textually that of the variable \mathbf{x} , as for example in: `A x; ...class A ...`

2.4.2 Class and interface declarations

The chapters 8.2 and 9 in [14] 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:

- $FieldDecl(\Gamma, C, v)$ indicates the nearest superclass of C (possibly C itself) which contains a declaration of the instance variable v and its declared type;
- $MethDecls(\Gamma, C, m)$ indicates all method declarations (*i.e.* both the class of the declaration and the signature) for method m in class C , or inherited from one of its superclasses, and not hidden by any of its superclasses;
- $MethSigs(\Gamma, C, m)$ returns all signatures for method m in class C , or inherited and not hidden by any of its superclasses.

Definition 3 For an environment Γ , containing a class declaration for C , *i.e.*

$\Gamma = \Gamma_1, C \text{ ext } C' \text{ impl } I_1, \dots, I_n \{v_1 : T_1, \dots, v_k : T_k, m_1 : MT_1, \dots, m_l : MT_l\}, \Gamma_2$, we define:

- $FieldDecl(\Gamma, Object, v) = \text{Undef}$ for any v and
 $FieldDecl(\Gamma, C, v) = (C, T_j)$ iff $v = v_j$ and
 $FieldDecl(\Gamma, C, v) = FieldDecl(\Gamma, C', v)$ iff $v \neq v_j \forall j \in \{1, \dots, k\}$
- $MethDecls(\Gamma, Object, m) = \emptyset$ and
 $MethDecls(\Gamma, C, m) = \{ (C, MT_j) \mid m = m_j \}$
 $\cup \{ (C'', MT'') \mid (C'', MT'') \in MethDecls(\Gamma, C', m), \text{ and}$
 $\forall j \in \{1, \dots, l\} : \text{if } m = m_j \text{ then } Args(MT_j) \neq Args(MT'') \}$
- $MethSigs(\Gamma, C, m) = \{ MT \mid \exists C'' \text{ with } (C'', MT) \in MethSigs(\Gamma, C, m) \}$

An example can be found in section 6.3. As for classes, we introduce some functions to look up the interface components: $MethDecls(\Gamma, I, m)$ indicates all method declarations (*i.e.* the interface of the declaration and the signature) for method m in class I , or inherited – and not hidden – from any of its superinterfaces; $MethSigs(\Gamma, I, m)$ returns all signatures for method m in interface I , or inherited – and not hidden – from a superinterface. Interface declarations are described in chapter 9 [14].

Definition 4 For an environment Γ , containing an interface declaration for I , *i.e.*

$\Gamma = \Gamma_1, I \text{ ext } I_1, \dots, I_n \{m_1 : MT_1, \dots, m_k : MT_k\}, \Gamma_2$ we define:

- $MethDecls(\Gamma, I, m) = \{ (I, MT_j) \mid m = m_j \} \cup \{ (I', MT') \mid$
 $\exists j \in \{1, \dots, n\} \text{ with } (I', MT') \in MethDecls(\Gamma, I_j, m)$
 $\text{and } \forall i \in \{1, \dots, k\} \text{ if } m = m_i \text{ then } Args(MT') \neq Args(MT_i) \}$
- $MethSigs(\Gamma, I, m) = \{ MT' \mid \exists I' : (I', MT') \in MethDecls(\Gamma, I, m) \}$

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

Lemma 2.1 For any environment Γ , type T , T' and identifiers v and m :

- $(T', MT) \in MethDecls(\Gamma, T, m) \implies$
 $-\Gamma \vdash T \sqsubseteq T' \text{ and } \Gamma(T') = T' \text{ ext } \dots \text{ impl } \dots \{ \dots m : MT \dots \} \text{ and}$
 $\forall C, T'' \text{ with: } \Gamma \vdash C \sqsubseteq T', \Gamma \vdash T \sqsubseteq C :$
 $\Gamma(C) \neq C \text{ ext } \dots \text{ impl } \dots \{ \dots m : Args(MT) \rightarrow T'' \}$
or

- $\Gamma \vdash T \leq T'$ and $\Gamma(T') = T' \text{ ext } \dots \{ \dots m : MT \dots \}$ and
 $\forall I, T'' : \text{with } \Gamma \vdash I \leq T', \Gamma \vdash T \leq I : \Gamma(I) \neq I \text{ ext } \dots \{ \dots m : \text{Args}(MT) \rightarrow T'' \}$
- $\text{FieldDecl}(\Gamma, C, v) = (C', T') \implies \Gamma(C') = C' \dots \{ \dots v : T \dots \}$ and $\Gamma \vdash C \sqsubseteq C'$ and
 $\forall C'', T'' \text{ with } \Gamma \vdash C \sqsubseteq C'', \Gamma \vdash C'' \sqsubseteq C' : \Gamma(C'') \neq C'' \text{ ext } \dots \text{impl} \dots \{ \dots v : T'' \}$

Proof 2.1 by induction on the number of class or interface declarations in Γ .

When a new class is declared as $C \text{ ext } C' \text{ impl } I_1, \dots, I_n \{ v_1 : T_1, \dots, v_k : T_k, m_1 : MT_1, \dots, m_l : T_l \}$, then [14] imposes the following requirements:

- there can be sequences of superinterfaces, instance variable declarations, and instance method declarations;
- the previous declarations should be well formed;
- there must not be a prior declaration of C
- there should be no cyclic subclass dependencies between C' and C
- C' should be a class – whose declaration might *follow* that of C (this is why we require $\Gamma \vdash C' \sqsubseteq C'$, rather than $\Gamma' \vdash C' \sqsubseteq C'$);
- the I_j should denote interfaces – whose declaration might also *follow* that of C ;
- the T_j should denote variable types – whose declaration might also *follow* that of C ;
- the MT_j should denote method types;
- no two instance variables should have the same identifier;
- no two instance methods should have the same identifier and 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 [19] instead of [14] 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 [19] instead of [14], 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.

$$\begin{array}{l}
n \geq 0, k \geq 0, l \geq 0 \\
\Gamma \vdash \Gamma' \diamond \\
\Gamma'(C) = \text{Undef} \\
\text{NOT } \Gamma \vdash C' \sqsubseteq C \\
\Gamma \vdash C' \sqsubseteq C' \\
\Gamma \vdash I_j \leq I_j \quad j \in \{1, \dots, n\} \\
\Gamma \vdash T_j \diamond_{\text{VarType}} \quad j \in \{1, \dots, k\} \\
\Gamma \vdash MT_j \diamond_{\text{MethType}} \quad j \in \{1, \dots, l\} \\
v_i = v_j \implies i = j \quad j, i \in \{1, \dots, k\} \\
m_i = m_j \implies i = j \text{ or } \text{Args}(MT_i) \neq \text{Args}(MT_j) \quad j, i \in \{1, \dots, l\} \\
\forall j \in \{1, \dots, l\} \quad MT \in \text{MethSigs}(\Gamma, C', m_j), \text{Args}(MT) = \text{Args}(MT_j) \implies \\
\Gamma \vdash \text{Res}(MT_j) <_{\text{wdn}} \text{Res}(MT) \\
\forall m, \forall j \in \{1, \dots, k\} \quad AT \rightarrow T \in \text{MethSigs}(\Gamma, I_j, m) \implies \\
\exists T' \text{ with } AT \rightarrow T' \in \text{MethSigs}(\Gamma, C, m), \Gamma \vdash T' <_{\text{wdn}} T \\
\hline
\Gamma \vdash \Gamma', C \text{ ext } C' \text{ impl } I_1, \dots, I_n \{ v_1 : T_1, \dots, v_k : T_k, m_1 : MT_1, \dots, m_l : T_l \} \diamond
\end{array}$$

When a new interface I is introduced as $I \text{ ext } I_1, \dots, I_n \{ m_1 : MT_1, \dots, m_n : T_1 \}$, then the following requirements must be satisfied:

- there may be sequences of superinterfaces and instance method declarations;
- the previous declarations should be well formed;
- there must not be a prior declaration of I ;
- there should be no cyclic subinterface dependencies between I and I_j ;
- the I_j should denote interfaces – whose declaration might also *follow* that of I ;
- the MT_j should denote method types;
- there shouldn't be two instance methods with the same identifier and same argument types;
- 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 [19] instead of [14].

$$\begin{array}{l}
n \geq 0, l \geq 0 \\
\Gamma \vdash \Gamma' \diamond \\
\Gamma'(I) = \text{Undef} \\
\text{NOT } \Gamma \vdash I_i \leq I \quad j \in \{1, \dots, n\} \\
\Gamma \vdash I_j \leq I_j \quad j \in \{1, \dots, n\} \\
\Gamma \vdash MT_j \diamond_{MethType} \quad j \in \{1, \dots, l\} \\
m_i = m_j \implies i = j \text{ or } \text{Args}(MT_i) \neq \text{Args}(MT_j) \\
MT \in \text{MethSigs}(\Gamma, I_i, m_j), \text{Args}(MT) = \text{Args}(MT_j) \implies \\
\Gamma \vdash \text{Res}(MT_j) <_{wdn} \text{Res}(MT) \quad \forall j \in \{1, \dots, k\}, i \in \{1, \dots, n\} \\
\hline
\Gamma \vdash \Gamma', I \text{ ext } I_1, \dots, I_n \{ m_1 : MT_1, \dots, m_l : T_l \} \diamond
\end{array}$$

2.5 Properties of well formed environments

Lemma 2.2 *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.*

Proof 2.2 *straightforward, by induction on the number of declarations in Γ .*

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 2.3 *If $\Gamma \vdash \Gamma \diamond$, then:*

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

- The \leq relationship is a partial order

Proof 2.3 straightforward; the first and second property can be shown by structural induction on the derivation of $\Gamma \vdash \mathbf{C} \sqsubseteq \mathbf{C}'$; the third and fifth property can be shown by induction on the number of declarations in Γ ; the fourth property can be shown by structural induction on the derivation of $\Gamma \vdash \mathbf{I} \leq \mathbf{I}'$.

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 another type, then the class is identical to the type, or the immediate superclass widens to that type, or the class implements a subinterface of the type; that if an interface widens to another type, then the interface is identical to the type, or one of the immediate superinterfaces is a subinterface of that type.

Lemma 2.4 If $\Gamma \vdash \Gamma \diamond$, then:

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

Proof 2.4 straightforward; the first property by structural induction on the derivation of $\Gamma \vdash \mathbf{I} <_{wdn} \mathbf{T}$; the second property by structural induction on the derivation of $\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{C}$; the third property by structural induction the derivation of $\Gamma \vdash \mathbf{C} <_{wdn} \mathbf{I}$; the fourth property by structural induction the derivation of $\Gamma \vdash \mathbf{I} <_{wdn} \mathbf{T}$; for the fifth property: reflexivity follows from the definition of $<_{wdn}$, and lemma 2.3, antisymmetry follows from the above properties and lemma 2.3, and transitivity follows by case analysis for $\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{T}'$, $\Gamma \vdash \mathbf{T}' <_{wdn} \mathbf{T}''$, the above properties, and lemma 2.3.

The following lemma says that all method/field identifiers declared in a type are inherited in any type that widens to it; that for any method identifier inherited from a superclass/superinterface or declared in a type \mathbf{T} , a method with same argument types and appropriate result type is declared in the type or in a superclass/superinterface if a method from a superclass/superinterface is hidden, then there exists another method with identical argument types, and a result type that widens to that of the hidden method.

Lemma 2.5 If $\Gamma \vdash \Gamma \diamond$, for types \mathbf{T} and \mathbf{T}' , with $\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{T}'$, and $\mathbf{MT} \in \text{MethSigs}(\Gamma, \mathbf{T}, \mathbf{m})$:

- $\exists_1 \mathbf{MT} \in \text{MethSigs}(\Gamma, \mathbf{T}', \mathbf{m})$ with $\text{Args}(\mathbf{MT}) = \text{Args}(\mathbf{MT}')$
- $\Gamma \vdash \text{Res}(\mathbf{MT}) <_{wdn} \text{Res}(\mathbf{MT}')$

Proof 2.5 using lemma 2.4 and the rules about well formed class and interface declarations, and induction on the number of declarations of Γ .

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} , the enriched Java_s

Some compile-time type information is used for the execution of Java method calls and of instance variable access. This information is calculated when type checking, and needs to be stored in the program before execution.

Therefore, we defined Java_{se} , an extended version of Java_s , which includes the appropriate type information. Furthermore, terms like $\text{addr}(i)$ represent references to objects, which will be necessary when, in the next chapter, we describe 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.

The syntax of Java_{se} may be obtained from the syntax of Java_s by applying the following modifications and additions:

$Expr$	$::=$	\dots	
		$Expr.[ArgType]MethName(Expr^*)$	instead of $Expr.MethName(Expr^*)$
		$Stmts$	
Var	$::=$	\dots	
		$Var.[ClassName]VarName$	instead of $Var.VarName$
		$\text{addr}(i)$	i an integer
$RefValue$	$::=$	\dots	
		$\text{addr}(i)$	i an integer

3.2 Variables, primitive values, and the null value

$$\frac{}{\vdash \text{null} : \text{nullT}}$$

$$Comp\{\text{null}, \Gamma\} = \text{null}$$

$$\frac{}{\vdash \text{true} : \text{bool}}$$

$$Comp\{\text{true}, \Gamma\} = \text{true}$$

$$\frac{}{\vdash \text{false} : \text{bool}}$$

$$Comp\{\text{false}, \Gamma\} = \text{false}$$

$$\frac{i \text{ is an integer}}{\vdash i : \text{int}}$$

$$Comp\{i, \Gamma\} = i$$

$$\frac{c \text{ is a character}}{\vdash c : \text{char}}$$

$$Comp\{c, \Gamma\} = c$$

$$\frac{\Gamma \vdash \mathbf{x} : \Gamma(\mathbf{x})}{\mathit{Comp}\{\{\mathbf{x}, \Gamma\}\} = \mathbf{x}}$$

3.3 Assignments, statement sequences, conditionals, return statements

An expression of type T' can be assigned to a variable of a type T , if T' can be widened to T .

$$\frac{\begin{array}{l} \Gamma \vdash \mathbf{v} : T \\ \Gamma \vdash \mathbf{e} : T' \\ \Gamma \vdash T' <_{\text{widen}} T \end{array}}{\Gamma \vdash \mathbf{v} := \mathbf{e} : \text{void}} \\ \mathit{Comp}\{\{\mathbf{v} := \mathbf{e}, \Gamma\}\} = \mathit{Comp}\{\{\mathbf{v}, \Gamma\}\} := \mathit{Comp}\{\{\mathbf{e}, \Gamma\}\}$$

$$\frac{\begin{array}{l} \Gamma \vdash \text{stmts} : T \\ \Gamma \vdash \text{stmts}' : T \\ \Gamma \vdash \mathbf{e} : \text{bool} \end{array}}{\Gamma \vdash (\text{if } \mathbf{e} \text{ then stmts else stmts}') : T} \\ \mathit{Comp}\{\{\text{if } \mathbf{e} \text{ then stmts else stmts}', \Gamma\}\} = \\ \text{if } \mathit{Comp}\{\{\mathbf{e}, \Gamma\}\} \text{ then } \mathit{Comp}\{\{\text{stmts}, \Gamma\}\} \text{ else } \mathit{Comp}\{\{\text{stmts}', \Gamma\}\}$$

$$\frac{\begin{array}{l} \Gamma \vdash \text{stmts} : T \\ \Gamma \vdash \text{stmt} : T' \end{array}}{\Gamma \vdash \text{stmts}; \text{stmt} : T'} \\ \mathit{Comp}\{\{\text{stmts} ; \text{stmt}, \Gamma\}\} = \mathit{Comp}\{\{\text{stmts}, \Gamma\}\} ; \mathit{Comp}\{\{\text{stmt}, \Gamma\}\}$$

$$\frac{\Gamma \vdash \text{return} : \text{void}}{\mathit{Comp}\{\{\text{return}, \Gamma\}\} = \text{return}}$$

$$\frac{\Gamma \vdash \mathbf{e} : T}{\Gamma \vdash \text{return } \mathbf{e} : \text{void}} \\ \mathit{Comp}\{\{\text{return } \mathbf{e}, \Gamma\}\} = \text{return } \mathit{Comp}\{\{\mathbf{e}, \Gamma\}\}$$

3.4 Field access

Note that only classes have fields

$$\frac{\begin{array}{l} \Gamma \vdash \text{var} : T \\ \mathit{FieldDecl}(\Gamma, T, \mathbf{v}) = (C, T') \neq \text{Undef} \end{array}}{\Gamma \vdash \text{var.v} : T'} \\ \mathit{Comp}\{\{\text{var.v}, \Gamma\}\} = \mathit{Comp}\{\{\text{var}, \Gamma\}\}.[C]\mathbf{v}$$

3.5 Method call

For method calls compare ch 15.11, [19]: 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, iff it is defined in a subclass or subinterface, and iff all argument types can be widened to from the argument types of the second 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 T and T_i , $i \in \{1, \dots, n + 1\}$, and identifier m , the most special declarations are defined as follows:

- $ApplMeths(\Gamma, \mathbf{m}, \mathbf{T}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n) = \{(\mathbf{T}', \mathbf{MT}') \mid (\mathbf{T}', \mathbf{MT}') \in MethDecls(\Gamma, \mathbf{T}, \mathbf{m})$
and $\mathbf{MT}' = \mathbf{T}'_1 \times \dots \times \mathbf{T}'_n \rightarrow \mathbf{T}'_{n+1}$ and $\Gamma \vdash \mathbf{T}_i <_{wdn} \mathbf{T}'_i$ for $i \in \{1, \dots, n\}\}^2$
- $(\mathbf{T}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n \rightarrow \mathbf{T}_{n+1})$ is more special than $(\mathbf{T}', \mathbf{T}'_1 \times \dots \times \mathbf{T}'_n \rightarrow \mathbf{T}'_{n+1})$ iff
 $\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{T}'$ and $\Gamma \vdash \mathbf{T}_i <_{wdn} \mathbf{T}'_i$ for all $i \in \{1, \dots, n\}$ ³
- $MostSpec(\Gamma, \mathbf{m}, \mathbf{T}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n) =$
 $\{(\mathbf{T}', \mathbf{MT}') \mid (\mathbf{T}', \mathbf{MT}') \in ApplMeths(\Gamma, \mathbf{m}, \mathbf{T}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n)$ and
if $(\mathbf{T}'', \mathbf{MT}'') \in ApplMeths(\Gamma, \mathbf{m}, \mathbf{T}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n)$ and $(\mathbf{T}'', \mathbf{MT}'')$ is more special than $(\mathbf{T}', \mathbf{MT}')$
then $\mathbf{T}'' = \mathbf{T}'$ and $\mathbf{MT}'' = \mathbf{MT}'\}$

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 [14], and the result type of the signature is the type of the message expression.

$$\frac{\Gamma \vdash \mathbf{e}_i : \mathbf{T}_i \quad i \in \{1, \dots, n\}, n \geq 1}{MostSpec(\Gamma, \mathbf{m}, \mathbf{T}_1, \mathbf{T}_2 \times \dots \times \mathbf{T}_n) = \{(\mathbf{T}, \mathbf{MT})\}}$$

$$\frac{\Gamma \vdash \mathbf{e}_1.\mathbf{m}(\mathbf{e}_2 \dots \mathbf{e}_n) : Res(\mathbf{MT})}{Comp\{\{\mathbf{e}_1.\mathbf{m}(\mathbf{e}_2 \dots \mathbf{e}_n), \Gamma\}\} =$$

$$Comp\{\{\mathbf{e}_1, \Gamma\}\}.\mathit{Args}(\mathbf{MT})\mathbf{m}(Comp\{\{\mathbf{e}_2, \Gamma\}\} \dots Comp\{\{\mathbf{e}_n, \Gamma\}\})$$

3.6 Method bodies

The next rule describes type checking of method bodies.

$$\frac{\begin{array}{l} \mathbf{mBody} = \mathbf{m} \text{ is } \lambda \mathbf{x}_1 : \mathbf{T}_1 \dots \lambda \mathbf{x}_n : \mathbf{T}_n. \{ \text{stmts} \} \\ x_i \neq \text{this} \quad i \in \{1, \dots, n\} \\ \mathbf{z}_1, \dots, \mathbf{z}_n \text{ are new variables in } \Gamma \\ \text{stmts}' = \text{stmts}[\mathbf{z}_1/\mathbf{x}_1, \dots, \mathbf{z}_n/\mathbf{x}_n] \\ \Gamma, \mathbf{x}_1 : \mathbf{T}_1 \dots \mathbf{x}_n : \mathbf{T}_n \vdash \text{stmts}' : \mathbf{T}' \\ \Gamma \vdash \mathbf{T}' <_{wdn} \mathbf{T} \end{array}}{\Gamma \vdash \mathbf{mbody} : \mathbf{T}_1 \times \dots \times \mathbf{T}_n \rightarrow \mathbf{T}}$$

$$Comp\{\{\mathbf{mBody}, \Gamma\}\} = \mathbf{m} \text{ is } \lambda \mathbf{x}_1 : \mathbf{T}_1 \dots \lambda \mathbf{x}_n : \mathbf{T}_n. \{ Comp\{\{\text{stmts}, \Gamma\}\} \}$$

The renaming of the variables in the method body (*i.e.* $\text{stmts}[\mathbf{z}_1/\mathbf{x}_1, \dots, \mathbf{z}_n/\mathbf{x}_n]$) is necessary in order to deal with name clashes between global or instance variable identifiers with formal parameter identifiers, and also, in order for the lemma 4.1 to hold – as pointed out in [23].

3.7 Class bodies

A class body \mathbf{cBody} satisfies its declaration $\Gamma(\mathbf{C})$ if it provides a method body for all the method declarations contained in $\Gamma(\mathbf{C})$.

$$\frac{\begin{array}{l} n \geq 0, k \geq 0, m \geq 0 \\ \Gamma \vdash \Gamma \diamond \\ \Gamma(\mathbf{C}) = \mathbf{C} \text{ ext } \mathbf{C}' \text{ impl } \mathbf{I}_1 \dots \mathbf{I}_n \{ \mathbf{v}_1 : \mathbf{T}_1 \dots \mathbf{v}_k : \mathbf{T}_k, \mathbf{m}_1 : \mathbf{MT}_1 \dots \mathbf{m}_l : \mathbf{MT}_l \} \\ \mathbf{cBody} = \mathbf{C} \text{ ext } \mathbf{C}' \{ \mathbf{mBody}_1, \dots, \mathbf{mBody}_l \} \\ \Gamma(\text{this}) = \text{Undef} \\ \mathbf{mBody}_i = \mathbf{m}_i \text{ is } \mathbf{mPrsSts}_i \quad i \in \{1, \dots, l\} \\ \Gamma, \text{this} : \mathbf{C} \vdash \mathbf{mBody}_i : \mathbf{MT}_i \quad i \in \{1, \dots, l\} \end{array}}{\Gamma \vdash \mathbf{cBody} : \Gamma(\mathbf{C})}$$

$$Comp\{\{\mathbf{cBody}, \Gamma\}\} = \mathbf{C} \text{ ext } \mathbf{C}' \{ Comp\{\{\mathbf{mBody}_1, \Gamma\}\} \dots Comp\{\{\mathbf{mBody}_l, \Gamma\}\} \}$$

²no requirement for the result type \mathbf{T}'_{n+1}

³again, no requirements for the result types \mathbf{T}_{n+1} and \mathbf{T}'_{n+1}

Note, that the method bodies \mathbf{mBody}_i are type checked in the environment $\Gamma, \mathbf{this} : \mathbf{C}$, which does not contain the instance variable declarations $\mathbf{v}_1 : \mathbf{T}_1 \dots \mathbf{v}_k : \mathbf{T}_k$. Thus, for instance variable access we force the use of the expression $\mathbf{this.v}_j$ as opposed to \mathbf{v}_j through the type system. The function $\mathit{MethBody}(\mathbf{m}, \mathbf{AT}, \mathbf{cBody})$ finds the method body with identifier \mathbf{m} and argument types same as \mathbf{AT} , in the class body \mathbf{cBody} – if any exists. It can easily be seen that because of the requirements for classes in 2.4.2, if the environment Γ is well formed, the function $\mathit{MethBody}(\mathbf{m}, \mathbf{AT}, \mathbf{cBody})$ returns an empty set or a set of one element.

Definition 6 For a class body $\mathbf{cBody} = \mathbf{C} \text{ ext } \mathbf{C}'\{ \mathbf{mBody}_1, \dots, \mathbf{mBody}_n \}$, defined in Γ as $\Gamma(\mathbf{C}) = \mathbf{C} \text{ ext } \mathbf{C}' \text{ impl } \dots \{ \mathbf{m}_1 : \mathbf{MT}_1 \dots \mathbf{m}_n : \mathbf{MT}_n \}$, we define:

$$\mathit{MethBody}(\mathbf{m}, \mathbf{AT}, \mathbf{cBody}) = \{ \mathbf{mPrsSts}_j \mid \mathbf{mBody}_j = \mathbf{m} \text{ is } \mathbf{mPrsSts}_j \text{ and } \mathit{Args}(\mathbf{MT}_j) = \mathbf{AT} \}$$

3.8 Programs

A program $\mathbf{p} = \{ \mathbf{cBody}_1, \dots, \mathbf{cBody}_n \}$ is well typed, if it contains a class body for each declared class, and if all the class bodies, \mathbf{cBody}_i , it consists of, are well typed and satisfy their declaration. Furthermore, each class is transformed by Comp .

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

The function $\mathit{MethBody}(\mathbf{m}, \mathbf{AT}, \mathbf{C}, \mathbf{p})$ finds the method body with identifier \mathbf{m} and argument types \mathbf{AT} , in the nearest superclass of class \mathbf{C} – if any exists. It returns a set of up to one tuple consisting of the class containing the appropriate method body, and the method body itself.

Definition 7 For a program $\mathbf{p} = \{ \mathbf{cBody}_1, \dots, \mathbf{cBody}_n \}$, we define:

$$\begin{array}{l} \mathit{MethBody}(\mathbf{m}, \mathbf{AT}, \mathbf{C}, \mathbf{p}) = \\ \quad \text{let } \mathbf{cBody} = \mathbf{C} \text{ ext } \mathbf{C}'\{ \dots \} \text{ in} \\ \quad \text{let } \mathbf{mBody} = \mathit{MethBody}(\mathbf{m}, \mathbf{AT}, \mathbf{cBody}) \text{ in} \\ \quad \text{if } \mathbf{mBody} = \emptyset \text{ then} \\ \quad \quad \text{if } \mathbf{C}' = \mathbf{Object} \text{ then } \emptyset \\ \quad \quad \text{else } \mathit{MethBody}(\mathbf{m}, \mathbf{AT}, \mathbf{C}', \mathbf{p}) \\ \quad \text{else } (\mathbf{C}, \mathbf{mBody}) \end{array}$$

3.9 Properties of the Java_s type system

The following lemma says, that if a class inherits a method declaration with method type: $\mathbf{m} : \mathbf{T}_1 \times \dots \times \mathbf{T}_n \rightarrow \mathbf{T}_{n+1}$ then it provides a method body that satisfies that method type.

Lemma 3.1 For any well formed environment Γ , variable types $\mathbf{T}, \mathbf{T}_1, \dots, \mathbf{T}_n, \mathbf{T}_{n+1}$, class \mathbf{C} and a Java_s program \mathbf{p} , If:

- $\Gamma \vdash \mathbf{p} : \Gamma$
- $(\mathbf{T}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n \rightarrow \mathbf{T}_{n+1}) \in \mathit{MethDecls}(\Gamma, \mathbf{C}, \mathbf{m})$

then

- $\Gamma \vdash \mathbf{C} \sqsubseteq \mathbf{T}$

- $\text{MethBody}(\mathbf{m}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n, \mathbf{C}, \mathbf{p}) = (\mathbf{T}, \lambda \mathbf{x}_1 : \mathbf{T}_1, \dots, \lambda \mathbf{x}_n : \mathbf{T}_n. \{ \text{stmts} \})$ and
 $\exists \mathbf{T}'_{n+1} : \Gamma, \text{this} : \mathbf{T}, \mathbf{x}_1 : \mathbf{T}_1, \dots, \mathbf{x}_n : \mathbf{T}_n \vdash \text{stmts} : \mathbf{T}_{n+1}$ and $\Gamma \vdash \mathbf{T}'_{n+1} <_{\text{wdn}} \mathbf{T}_{n+1}$

Proof 3.1 Because of lemma 2.1, we know that \mathbf{T} is a class, $\Gamma \vdash \mathbf{C} \sqsubseteq \mathbf{T}$, and

$$\Gamma(\mathbf{T}) = \mathbf{T} \text{ ext } \dots \text{ impl } \dots \{ \dots \mathbf{m} : \mathbf{T}_1 \times \dots \times \mathbf{T}_n \rightarrow \mathbf{T}_{n+1} \dots \}.$$

We now apply the rule about well typed class bodies (section 3.7) “backwards” – this is legal, because the type system satisfies the subformula property – and we obtain that the class body for \mathbf{T} has the form: $\mathbf{T} \text{ ext } \dots \{ \dots \mathbf{m} \text{ is } \lambda \mathbf{x}_1 : \mathbf{T}_1 \dots \lambda \mathbf{x}_n : \mathbf{T}_n. \{ \text{stmts} \} \dots \}$.

Therefore, $\text{MethBody}(\mathbf{m}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n, \mathbf{T}, \mathbf{p}) = (\mathbf{T}, \lambda \mathbf{x}_1 : \mathbf{T}_1, \dots, \lambda \mathbf{x}_n : \mathbf{T}_n. \{ \text{stmts} \})$. Additionally, by a further “backwards” application of the type rule in 3.6, we obtain that:

$$\Gamma, \text{this} : \mathbf{T}, \mathbf{x}_1 : \mathbf{T}_1, \dots, \mathbf{x}_n : \mathbf{T}_n \vdash \text{stmts} : \mathbf{T}'_{n+1}, \text{ for a type } \mathbf{T}'_{n+1} \text{ with } \Gamma \vdash \mathbf{T}'_{n+1} <_{\text{wdn}} \mathbf{T}_{n+1}.$$

It only remains to be shown that:

$$\text{MethBody}(\mathbf{m}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n, \mathbf{T}, \mathbf{p}) = \text{MethBody}(\mathbf{m}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n, \mathbf{C}, \mathbf{p}).$$

Assuming that the equality did not hold, we would obtain that a class \mathbf{C}' , which is a superclass of \mathbf{C} , and not a superclass of \mathbf{T} , contains a method body for \mathbf{m} with argument types $\mathbf{T}_1 \times \dots \times \mathbf{T}_n$. By lemma 2.3 we then obtain that \mathbf{C}' is a subclass of \mathbf{T} , and therefore, with lemma 2.1 we obtain that $(\mathbf{T}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n) \rightarrow \mathbf{T}_{n+1} \notin \text{MethDecls}(\Gamma, \mathbf{T}, \mathbf{m})$ which gives a contradiction.

We can now formulate the more general lemma, which says that in a well typed Java_s program any class that widens to another superclass or superinterface provides an implementation for each method exported by the superclass or superinterface.

Lemma 3.2 For any well formed environment Γ , variable types $\mathbf{T}, \mathbf{T}_1, \dots, \mathbf{T}_n, \mathbf{T}_{n+1}$, class \mathbf{C} and a Java_s program \mathbf{p} , if:

- $\Gamma \vdash \mathbf{p} : \Gamma$
- $\Gamma \vdash \mathbf{C} <_{\text{wdn}} \mathbf{T}$
- $\mathbf{T}_1 \times \dots \times \mathbf{T}_n \rightarrow \mathbf{T}_{n+1} \in \text{MethSigs}(\Gamma, \mathbf{T}, \mathbf{m})$

then

- $\exists \mathbf{T}'_{n+1}, \mathbf{C}' : (\mathbf{C}', \mathbf{T}_1 \times \dots \times \mathbf{T}_n \rightarrow \mathbf{T}'_{n+1}) \in \text{MethDecls}(\Gamma, \mathbf{C}, \mathbf{m})$, and
 $\Gamma \vdash \mathbf{T}'_{n+1} <_{\text{wdn}} \mathbf{T}_{n+1}$ and $\Gamma \vdash \mathbf{C} \sqsubseteq \mathbf{C}'$ and
- $\text{MethBody}(\mathbf{m}, \mathbf{T}_1 \times \dots \times \mathbf{T}_n, \mathbf{p}, \mathbf{C}) = (\mathbf{C}', \lambda \mathbf{x}_1 : \mathbf{T}_1, \dots, \lambda \mathbf{x}_n : \mathbf{T}_n. \{ \text{stmts} \})$ and
 $\Gamma, \text{this} : \mathbf{C}', \mathbf{x}_1 : \mathbf{T}_1, \dots, \mathbf{x}_n : \mathbf{T}_n \vdash \text{stmts} : \mathbf{T}''_{n+1}$ and $\Gamma \vdash \mathbf{T}''_{n+1} <_{\text{wdn}} \mathbf{T}'_{n+1}$

Proof 3.2 If \mathbf{T} is a class, then $\Gamma \vdash \mathbf{C} \sqsubseteq \mathbf{T}$, and by lemma 2.1, there exists a \mathbf{C}' , with $\Gamma \vdash \mathbf{C} \sqsubseteq \mathbf{C}'$ and $(\mathbf{C}', \mathbf{T}_1 \times \dots \times \mathbf{T}_n) \in \text{MethDecls}(\Gamma, \mathbf{C}, \mathbf{m})$. The rest follows from lemma 3.1. If \mathbf{T} is an interface, then because of lemma 2.4, there exist a class \mathbf{C}'' and an interface \mathbf{I}' , with $\Gamma \vdash \mathbf{C} \sqsubseteq \mathbf{C}''$, $\Gamma \vdash \mathbf{C}'' \text{ :impl } \mathbf{I}'$ and $\Gamma \vdash \mathbf{I}' \leq \mathbf{I}$. By lemma 2.1 and the rules about well formed class declarations, we obtain that there exists a \mathbf{T}''_{n+1} , with $(\mathbf{C}'', \mathbf{T}_1 \times \dots \times \mathbf{T}_n \rightarrow \mathbf{T}''_{n+1}) \in \text{MethDecls}(\Gamma, \mathbf{C}'', \mathbf{m})$, $\Gamma \vdash \mathbf{T}''_{n+1} <_{\text{wdn}} \mathbf{T}_{n+1}$. Again by application of lemma 2.1, we obtain that there exist \mathbf{C}' , \mathbf{T}'_{n+1} , with $(\mathbf{C}', \mathbf{T}_1 \times \dots \times \mathbf{T}_n \rightarrow \mathbf{T}'_{n+1}) \in \text{MethDecls}(\Gamma, \mathbf{C}, \mathbf{m})$, $\Gamma \vdash \mathbf{C} \sqsubseteq \mathbf{C}'$, and $\Gamma \vdash \mathbf{T}'_{n+1} <_{\text{wdn}} \mathbf{T}''_{n+1}$. The rest follows as in the previous case.

3.9.1 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 \mathbf{T} , also has type \mathbf{T}' if \mathbf{T} is a subtype of

T' . In the case of Java, where subtypes are expressed by the $<_{wdn}$ relation, it would have the form:

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

The following Java program justifies the absence of the subsumption rule:

```
class A { ... }
class A1 extend A { ... }
class B {
  char f(A x){ ... }
  int f(A1 y){ ... }
  void g(A1 z) { ...f(z) ... }
  void h(A u, A1 v) { ...f(u) ... }
}
```

The type of $f(z)$ in the body of g is `int`. Namely, the parameter z is of type `A1`, both functions f are applicable, but the second function, declared as `int f(A1 y)` is more specific than the first function, declared as `char f(A x)`. Thus, the type of $f(z)$ is the same as the result type of the second function, *i.e.* `int`. However, if we had a subsumption rule, then the type of $f(z)$ could also be `char`. Namely, by virtue of the subsumption rule, the type of z could be `A`. Then only the first function f would be applicable, and the type of $f(z)$ would be the same as the result type of the first function, *i.e.* `char`. In fact, introduction of the subsumption rule would make this type system non-deterministic – although [7] 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.

3.10 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 `addr(i)`.

The type of references 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 Java_{se} terms depends on both the environment *and* the state, and type assertions for Java_{se} terms τ have the form $\Gamma, \sigma \vdash \tau : T$.

3.10.1 References

$$\frac{\sigma(\text{addr}(i)) = \langle\langle \dots \rangle\rangle^C}{\Gamma, \sigma \vdash \text{addr}(i) : C}$$

3.10.2 Field access

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.* C) instead of the type of the variable at the left of the field access (*i.e.* T).

$$\frac{\Gamma, \sigma \vdash \text{var} : T \quad \Gamma, \sigma \vdash T <_{wdn} C \quad \text{FieldDecl}(\Gamma, C, v) = (C, T') \neq \text{Undef}}{\Gamma, \sigma \vdash \text{var}.[C]v : T'}$$

3.10.3 Method call

In Java_{se} method calls we search for appropriate methods, using the descriptor signature $(T_2 \times \dots \times T_n)$, instead of the types of the actual expressions (T'_2, \dots, 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. This is expressed in

Definition 8 *Given environment Γ , types T_1, \dots, T_n , argument types $AT = T_2 \times \dots \times T_n$ and an identifier m , we define:*

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

Using this definition:

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

Lemma 3.3 *For a well formed environment Γ , types $T'_1, T_1 \dots T_n$, argument types, $AT = T_2 \times \dots \times T_n$, where $\Gamma \vdash T_1 <_{wdn} T'_1$:*

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

Proof 3.3 *applying the definitions, and lemma 2.5.*

3.11 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:

Lemma 3.4 *For any environment Γ , for any state σ , Java_s term τ , type T , it holds that:*

$$\Gamma \vdash \tau : T \implies \Gamma, \sigma \vdash \mathit{Comp}\{\tau, \Gamma\} : T$$

Proof 3.4 *First argue, that for any Java_s term τ , its Java_{se} enriched form, $\mathit{Comp}\{\tau, \Gamma\}$ does not contain any references (i.e. a $\mathit{addr}(i)$) as a subterm. Then by structural induction on the Java_s derivation of $\Gamma \vdash \tau : T$, applying lemmas 2.5, 3.3 and 2.1.*

The type system assigns unique types to any well typed Java_s or Java_{se} term:

Lemma 3.5 *For types T, T' , state σ , environment Γ , a Java_s term τ , or a Java_{se} term τ' :*

- $\Gamma \vdash \tau : T \text{ and } \Gamma \vdash \tau : T' \implies T = T'$.
- $\Gamma, \sigma \vdash \tau' : T \text{ and } \Gamma, \sigma \vdash \tau' : T' \implies T = T'$.

Proof 3.5 *by structural induction over the derivation of $\Gamma \vdash \tau : T$, or of $\Gamma, \sigma \vdash \tau' : T$*

3.11.1 The substitution property

Java, and therefore Java_s too, does not have a “substitution property”. The *substitution property* states that replacing a subexpression by a new subexpression of a subtype of the type of the original subexpression does not affect the type of the enclosing expression – up to the widening relationship.

The example from 3.9.1 illustrates why Java does not have this property: The type of $f(u)$ in the body of h is `char`. However, if we replace the subexpression u of type A by v of type $A1$, ($A1$ widens to A), we obtain $f(v)$ which has type `int`, and `int` *does not* widen to `char`.

Java_s expressions translated to the enriched language, have this property, as stated in the following lemma. However, the lemma does not apply to general Java_{se} terms. For example, for classes C, C' , state σ and environment Γ , such that $\Gamma \vdash C \sqsubseteq C'$, it holds that $\Gamma, x : C', y : C', z : C, \sigma \vdash x := y : \text{void}$. But, if we replace x by z , we obtain $z := y$ which is type incorrect. The property does not even apply to Java_{se} expressions, because – as pointed out by [23] – Java_{se} expressions may include lists of statements.

Lemma 3.6 *For any environment Γ , state σ , types T, T', T'' , Java_s expressions e and e'' , Java_{se} expressions e' and e''' , with $\text{Comp}\{e, \Gamma\} = e'$, $\text{Comp}\{e'', \Gamma\} = e'''$, if $\Gamma, x : T' \vdash e : T$, and $\Gamma \vdash T'' <_{\text{wdn}} T'$, and $\Gamma \vdash e'' : T''$, then, there exists a T''' with:*

$$\Gamma, \sigma \vdash [e'''/x]e' : T''' \text{ and } \Gamma, \sigma \vdash T''' <_{\text{wdn}} T$$

Proof 3.6 *by structural induction on the derivation of $\Gamma, x : T' \vdash e : T$, using lemma 3.4, using the Java_{se} type rules from this section, and applying lemmas 2.1, 2.5, 3.3. Note, that because e, e'' are Java_s expressions, they do not contain assignments as subterms, and therefore neither do the Java_{se} expressions e', e''' contain assignments as subterms.*

4 The operational semantics

For the operational semantics we need a notion of state. The state is flat, and it consists of mappings from identifiers to primitive values or to references, and from references to objects.

$$\text{state} ::= (\text{Ident} \longrightarrow (\text{Value}))^* \cup (\text{RefValue} \longrightarrow \text{Object})^*$$

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,[14], ch 9.5

$$\text{Object} ::= \langle\langle (\text{LabelName } \text{ClassName} : \text{Value})^* \rangle\rangle^{\text{ClassName}}$$

For example, as in section 6: $\langle\langle x \text{ C1: } -3, u \text{ C2: } 'b', x \text{ C3: } \text{addr}(4) \rangle\rangle^{\text{C3}}$ is an object of class C3 . It inherits the field x from C1 , a superclass of C3 , and the field u from C2 . Furthermore, it has the field x which is reference to the object stored at $\text{addr}(4)$.

A configuration is tuple of a Java_{se} term and a state, or just a state.

$$\text{configuration} ::= \langle \text{Java}_{se}\text{-term}, \text{state} \rangle \cup \langle \text{state} \rangle$$

The operational semantics is a mapping from programs and configurations to configurations.

$$\rightsquigarrow : \text{Java}_{se}\text{program} \longrightarrow \text{configuration} \longrightarrow \text{configuration}$$

For a given program p , we also have:

$$\rightsquigarrow_p : \text{configuration} \longrightarrow \text{configuration}$$

First, 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, and variables to contain values according to their type.

Definition 9 A value \mathbf{val} weakly conforms to a type \mathbf{T} in an environment Γ and a state σ iff:

- if \mathbf{val} is a primitive value, then \mathbf{T} is a primitive type, and $\mathbf{val} \in \mathbf{T}$
- if $\mathbf{val} = \mathbf{null}$, then \mathbf{T} is an interface or class
- if $\mathbf{val} = \mathbf{addr}(j)$, then there exists a class \mathbf{C} with: $\Gamma \vdash \mathbf{C} <_{wdn} \mathbf{T}$, and $\sigma(\mathbf{addr}(j)) = \langle \langle \dots \rangle \rangle^{\mathbf{C}}$.

A value \mathbf{val} conforms to a type \mathbf{T} in an environment Γ and a state σ iff:

- \mathbf{val} weakly conforms to \mathbf{T} in Γ and σ
and
- if $\mathbf{val} = \mathbf{addr}(j)$, then $\sigma(\mathbf{addr}(j)) = \langle \langle \mathbf{v}_1 \mathbf{C}_1 : \mathbf{val}_1, \dots, \mathbf{v}_n \mathbf{C}_n : \mathbf{val}_n \rangle \rangle^{\mathbf{C}}$, for a class \mathbf{C} , and for all labels \mathbf{v} , classes \mathbf{C}' , types \mathbf{T}' with $(\mathbf{C}', \mathbf{T}') \in \mathcal{F}ieldDecl(\Gamma, \mathbf{C}, \mathbf{v})$, there exists a $\mathbf{k} \in \{1, \dots, n\}$ such that $\mathbf{v}_{\mathbf{k}} = \mathbf{v}$, $\mathbf{C}_{\mathbf{k}} = \mathbf{C}'$, and $\mathbf{val}_{\mathbf{k}}$ weakly conforms to \mathbf{T}' in Γ and σ .

Furthermore, a state σ conforms to an environment Γ iff

- for any identifier \mathbf{x} , if $\Gamma(\mathbf{x}) \neq \mathbf{Undef}$ then $\sigma(\mathbf{x})$ conforms to $\Gamma(\mathbf{x})$ in Γ and σ .
- for any interger \mathbf{i} , if $\sigma(\mathbf{addr}(\mathbf{i})) = \langle \langle \dots \rangle \rangle^{\mathbf{C}}$, then $\mathbf{addr}(\mathbf{i})$ conforms to \mathbf{C} in Γ and σ .

Also, an environment Γ conforms to environment Γ' iff

- for any identifier \mathbf{x} , if $\Gamma'(\mathbf{x}) \neq \mathbf{Undef}$, then $\Gamma(\mathbf{x}) = \Gamma'(\mathbf{x})$, and
- for any identifier \mathbf{x} , if $\Gamma'(\mathbf{x}) = \mathbf{Undef} \neq \Gamma(\mathbf{x})$, then \mathbf{x} is declared in Γ as a variable.⁴

Lemma 4.1 Given two environments Γ, Γ' , where Γ conforms to Γ' ,

- $\Gamma \vdash \Gamma \diamond \implies \Gamma' \vdash \Gamma' \diamond$
- for any program \mathbf{p} : $\Gamma' \vdash \mathbf{p} : \Gamma' \implies \Gamma \vdash \mathbf{p} : \Gamma$
- for any term \mathbf{t} , and type \mathbf{T} : $\Gamma' \vdash \mathbf{t} : \mathbf{T} \implies \Gamma \vdash \mathbf{t} : \mathbf{T}$
- $\Gamma \vdash \mathbf{T} <_{wdn} \mathbf{T}' \iff \Gamma' \vdash \mathbf{T} <_{wdn} \mathbf{T}'$
- For any $\mathbf{T}_1, \dots, \mathbf{T}_n$: $FirstFit(\Gamma, \mathbf{m}, \mathbf{T}_1, \mathbf{T}_2 \times \dots \times \mathbf{T}_n) = FirstFit(\Gamma', \mathbf{m}, \mathbf{T}_1, \mathbf{T}_2 \times \dots \times \mathbf{T}_n)$

Proof 4.1 straightforward application of the definitions

Definition 10 For an object $\mathbf{obj} = \langle \langle \mathbf{l}_1 \mathbf{C}_1 : \mathbf{val}_1, \mathbf{l}_2 \mathbf{C}_2 : \mathbf{val}_2, \dots, \mathbf{l}_n \mathbf{C}_n : \mathbf{val}_n \rangle \rangle^{\mathbf{C}'}$ we define $\mathbf{obj}(\mathbf{f}, \mathbf{C})$, the access to field \mathbf{f} declared in class \mathbf{C} as

$$\mathbf{obj}(\mathbf{f}, \mathbf{C}) = \mathbf{val}_i \text{ where } \mathbf{f} = \mathbf{l}_i \text{ and } \mathbf{C} = \mathbf{C}_i$$

⁵ We also define access to component \mathbf{f} , \mathbf{C} of an object stored at a reference \mathbf{z} in state σ , as:

$$\sigma(\mathbf{z}, \mathbf{f}, \mathbf{C}) = \sigma(\mathbf{z})(\mathbf{f}, \mathbf{C})$$

For a state σ , a value \mathbf{val} , an identifier or reference \mathbf{z} , an object \mathbf{obj} , a class \mathbf{C} , and a variable \mathbf{f} , we define the following state and object updates:

⁴In other words, Γ may not introduce any new classes or interfaces

⁵Notice, that if \mathbf{obj} conforms to \mathbf{C} , and \mathbf{i} exists such that $\mathbf{f} = \mathbf{l}_i$, then it is unique.

- A new state: $\sigma[z \mapsto \text{val}]$, such that:

$$\begin{aligned} \sigma[z \mapsto \text{val}](z) &= \text{val} && \text{and} \\ \sigma[z \mapsto \text{val}](z') &= \sigma(z') && \text{for } z' \neq z : \end{aligned}$$

- A new object: $\text{obj}[f, C \mapsto \text{val}]$, such that:

$$\begin{aligned} \text{obj}[f, C \mapsto \text{val}](f, C) &= \text{val} && \text{and} \\ \text{obj}[f, C \mapsto \text{val}](f', C') &= \text{obj}(f', C') && f \neq f' \text{ or } C \neq C' \end{aligned}$$

- A new state: $\sigma[z, f, C \mapsto \text{val}]$, such that:

$$\sigma[z, f, C \mapsto \text{val}] = \sigma[z \mapsto \sigma(z)][f, C \mapsto \text{val}]$$

We distinguish ground terms which cannot be further rewritten, and l-ground terms, which are “almost ground”, but 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 $t = \text{addr}(i)$, for some i .
- l-ground if t is ground, or $t = \text{ident}$ for some identifier ident , or if $t = \text{addr}(i).[C]v$ for a class C and a field v

4.2 Statement sequences, conditionals and return statements

$$\frac{\langle \text{stmts}, \sigma \rangle \rightsquigarrow_p \langle \sigma' \rangle}{\langle \text{stmts}; \text{stmt}, \sigma \rangle \rightsquigarrow_p \langle \text{stmt}, \sigma' \rangle}$$

$$\frac{\langle \text{stmts}, \sigma \rangle \rightsquigarrow_p \langle \text{stmts}', \sigma' \rangle}{\langle \text{stmts}; \text{stmt}, \sigma \rangle \rightsquigarrow_p \langle \text{stmts}'; \text{stmt}, \sigma' \rangle}$$

$$\frac{\langle \text{expr}, \sigma \rangle \rightsquigarrow_p \langle \text{expr}', \sigma' \rangle}{\langle \text{if expr then stmts else stmts}', \sigma \rangle \rightsquigarrow_p \langle \text{if expr}' then stmts else stmts}', \sigma' \rangle}$$

$$\frac{}{\langle \text{if true then stmts else stmts}', \sigma \rangle \rightsquigarrow_p \langle \text{stmts}, \sigma \rangle}$$

$$\frac{}{\langle \text{if false then stmts else stmts}', \sigma \rangle \rightsquigarrow_p \langle \text{stmts}', \sigma \rangle}$$

$$\frac{}{\langle \text{return}, \sigma \rangle \rightsquigarrow_p \langle \sigma \rangle}$$

$$\frac{\langle \text{expr}, \sigma \rangle \rightsquigarrow_p \langle \text{expr}', \sigma' \rangle}{\langle \text{return expr}, \sigma \rangle \rightsquigarrow_p \langle \text{return expr}', \sigma' \rangle}$$

$$\frac{\text{val} \quad \text{is ground}}{\langle \text{return val}, \sigma \rangle \rightsquigarrow_p \langle \text{val}, \sigma \rangle}$$

4.3 Variables

$$\frac{}{\langle \text{ident}, \sigma \rangle \rightsquigarrow_p \langle \sigma(\text{ident}), \sigma \rangle}$$

$$\frac{}{\langle \text{addr}(\mathbf{i}).[\mathbf{C}]\mathbf{v}, \sigma \rangle \rightsquigarrow_p \langle \sigma(\text{addr}(\mathbf{i}), \mathbf{v}, \mathbf{C}), \sigma \rangle}$$

$$\frac{\langle \text{var}, \sigma \rangle \rightsquigarrow_p \langle \text{var}', \sigma \rangle}{\langle \text{var}.[\mathbf{C}]\mathbf{v}, \sigma \rangle \rightsquigarrow_p \langle \text{var}'.[\mathbf{C}]\mathbf{v}, \sigma \rangle}$$

The rules about assignment in 4.4 will prevent an expression like \mathbf{x} or $\text{addr}(\mathbf{i}).[\mathbf{C}]\mathbf{v}$ from being rewritten any further if it is the left hand side of an assignment. They would allow an expression of the form $\mathbf{u}[\mathbf{C1}].\mathbf{w}[\mathbf{C2}].\mathbf{x}[\mathbf{C3}].\mathbf{y}$ to be rewritten to an expression of the form $\text{addr}(\mathbf{j})[\mathbf{C3}].\mathbf{y}$ for some \mathbf{j} . Furthermore, there is *no* rule of the form $\langle \text{addr}(\mathbf{j}), \sigma \rangle \rightsquigarrow_p \langle \sigma(\text{addr}(\mathbf{j})), \sigma \rangle$. This is so, 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.

4.4 Assignment

$$\frac{\langle \text{expr}, \sigma \rangle \rightsquigarrow_p \langle \text{expr}', \sigma' \rangle}{\langle \text{var} := \text{expr}, \sigma \rangle \rightsquigarrow_p \langle \text{var} := \text{expr}', \sigma' \rangle}$$

val is ground
 var is not l-ground

$$\frac{\langle \text{var}, \sigma \rangle \rightsquigarrow_p \langle \text{var}', \sigma' \rangle}{\langle \text{var} := \text{val}, \sigma \rangle \rightsquigarrow_p \langle \text{var}' := \text{val}, \sigma' \rangle}$$

val is ground
 id is an identifier

$$\frac{}{\langle \text{id} := \text{val}, \sigma \rangle \rightsquigarrow_p \langle \sigma[\text{id} \mapsto \text{val}], \sigma \rangle}$$

val is ground

$$\frac{}{\langle \text{addr}(\mathbf{i}).[\mathbf{C}]\mathbf{v} := \text{val}, \sigma \rangle \rightsquigarrow_p \langle \sigma[\text{addr}(\mathbf{i}), \mathbf{v}, \mathbf{C} \mapsto \text{val}], \sigma \rangle}$$

Note that we have *no* rule of the form $\langle \text{addr}(\mathbf{j}) := \text{value}, \sigma \rangle \rightsquigarrow_p \dots$. This is so, because in Java overwriting of objects is not possible – only sending messages to them, or overwriting selected instance variables.

4.5 Method call

In method calls expressions are evaluated left to right, cf ch 9.3 in [19]. The first rule describes rewiring the k^{th} expression, where all the previous expressions (*i.e.* $\mathbf{e}_i, i \in \{1, \dots, k-1\}$) are ground. The second rule describes dynamic method look up, taking into account the argument types, and the statically calculated method descriptor:

$$\frac{\mathbf{e}_i \text{ is ground, for all } i \in \{1, \dots, k-1\}, n \geq k \geq 1}{\langle \mathbf{e}_k, \sigma \rangle \rightsquigarrow_p \langle \mathbf{e}'_k, \sigma' \rangle}$$

$$\frac{}{\langle \mathbf{e}_1.[\text{AT}]m(\mathbf{e}_2, \dots, \mathbf{e}_k, \dots, \mathbf{e}_n), \sigma \rangle \rightsquigarrow_p \langle \mathbf{e}_1.[\text{AT}]m(\mathbf{e}_2, \dots, \mathbf{e}'_k, \dots, \mathbf{e}_n), \sigma' \rangle}$$

$$\begin{array}{l}
\text{val}_i \quad \text{is ground } i \in \{1, \dots, n\}, n \geq 1 \\
\sigma(\text{val}_1) = \langle \langle \dots \rangle \rangle^c \\
\text{AT} = \text{T}_2 \times \dots \times \text{T}_n \\
\text{MethBody}(m, \text{AT}, \text{C}, p) = (\text{C}', \lambda x_2 : \text{T}_2 \dots \lambda x_n : \text{T}_n. \{ \text{stmts} \}) \\
z_i \text{ are new identifiers in } \sigma \\
\sigma' = \sigma[z_1 \mapsto \text{val}_1] \dots [z_n \mapsto \text{val}_n] \\
\text{stmts}' = \text{stmts}[z_1/\text{this}, z_2/x_2, \dots, z_n/x_n] \\
\hline
\langle \text{val}_1.\text{[AT]m}(\text{val}_2, \dots, \text{val}_n), \sigma \rangle \rightsquigarrow_p \langle \text{stmts}', \sigma' \rangle
\end{array}$$

where, $\tau[t'/x]$ has the usual meaning of replacing the variable x by the term t' in the term τ .

4.6 Properties of the operational semantics

The operational semantics is deterministic:

Lemma 4.2 *For any configuration with a state that conforms to the environment, and any Java_{se} term, the relation \rightsquigarrow_p determines at most one step.*

Proof 4.2 *by case analysis over the structure of the Java_{se} term*

Rewriting variables on the left hand sides of assignments does not make their type more special:

Lemma 4.3 *For any Java_{se} variables var, var' . If $\langle \text{var}, \sigma \rangle \rightsquigarrow_p \langle \text{var}', \sigma \rangle$, and $\Gamma \vdash \text{var} : \text{T}$, and var is not l-ground, then*

$\Gamma \vdash \text{var}' : \text{T}$, and var' is not ground.

Proof 4.3 *We show that $\Gamma \vdash \text{var}' : \text{T}$ by structural induction over the rewrite of $\langle \text{var}, \sigma \rangle \rightsquigarrow_p \langle \text{var}', \sigma \rangle$. Because var is not l-ground, $\text{var} = \text{var}''.\text{[C]v}$, $\text{var}'' \neq \text{addr}(\dots)$. Therefore, $\text{var}' = \text{var}'''\text{[C]v}$, where $\langle \text{var}''', \sigma \rangle \rightsquigarrow_p \langle \text{var}''', \sigma \rangle$. This proves that var' is not ground.*

Lemma 4.4 *For any Java_{se} term τ , state σ and environment Γ , if τ does not contain as assignment to $\text{addr}(\dots)$ as subterm, and $\langle \text{Comp}\{\tau, \Gamma\}, \sigma \rangle \rightsquigarrow_p \langle \tau', \sigma \rangle$, then τ' does not contain as assignment to $\text{addr}(\dots)$ as subterm.*

Proof 4.4 *by structural induction over τ , and then analysis over applicable rewriting rules.*

5 Soundness of the Java_s type system

Theorem 1 Subject Reduction *For a state σ that conforms to an environment Γ , a Java_{se} program p with $\Gamma \vdash p : \Gamma$, a non-ground Java_{se} term τ that does not contain any assignment of the form $\text{addr}(i) := \dots$, and type T with $\Gamma, \sigma \vdash \tau : \text{T}$, there exist σ', τ' such that:*

- $\langle \tau, \sigma \rangle \rightsquigarrow_p \langle \tau', \sigma' \rangle$, and
 - τ' contains the subterm $\text{null}.\text{[C]v}$ or $\text{null}.\text{[AT]m}(v_1, \dots, v_n)$
 - or
 - $\exists \Gamma', \text{T}' : \Gamma'$ conforms to Γ , σ' conforms to Γ' , and $\Gamma', \sigma' \vdash \tau' : \text{T}'$, and $\Gamma \vdash \text{T} <_{wdn} \text{T}'$

or

- $\langle \tau, \sigma \rangle \rightsquigarrow_p \langle \sigma' \rangle$ and σ' conforms to Γ

Proof by structural induction over the derivation of $\Gamma, \sigma \vdash \tau : T$. The interesting cases are:

1. Last step was an assignment, *i.e.* $\tau = \text{var} := \text{exp}$, and therefore $T = \text{void}$, and there exist T'', T''' , with $\Gamma, \sigma \vdash \text{var} : T''$, and $\Gamma, \text{sigma} \vdash \text{exp} : T'''$ and $\Gamma \vdash T''' <_{\text{wdn}} T''$. There are the following three cases:
 - (a) exp is not ground: by applying the induction hypothesis, there exist exp', σ' with $< \text{exp}, \sigma > \rightsquigarrow_p < \text{exp}', \sigma' >$. Therefore $< \tau, \sigma > \rightsquigarrow_p < \text{var} := \text{exp}', \sigma' >$
 - i. exp' contains a `null` access as a subexpression. This completes the case.
 - ii. There exist Γ'', T'''' with Γ'' conforms to Γ' , σ' conforms to Γ'' , and $\Gamma'', \sigma' \vdash \text{exp}' : T''''$. Therefore, $\Gamma'' \vdash T'''' <_{\text{wdn}} T'$, and therefore: $\Gamma'', \sigma' \vdash \text{var} := \text{exp}' : \text{void}$. This completes the lemma, with $T' = T = \text{void}$, $\Gamma' = \Gamma''$ and $\tau' = \text{var} := \text{exp}'$.
 - (b) exp is ground, and var is not l-ground: similar argumentation with the previous case, also using lemma 4.3.
 - (c) exp is ground, and var is l-ground: If $\text{var} = \text{ident}$, then because $\Gamma \vdash T''' <_{\text{wdn}} T''$, if we take $\sigma' = \sigma[\text{ident} \mapsto \text{exp}]$, conforms to $\Gamma' = \Gamma$, and the lemma is proven. Similar argument for the case where $\text{var} = \text{addr}(i).[C]v$. No other case because of lemma 4.4.
2. Last step was a method call. Then $\tau = e_1.[\text{AT}]m(e_2 \dots e_n)$, and there exist $T_1, \dots, T_n, T'_1, \dots, T'_n, MT'', T''$ with $\Gamma, \sigma \vdash e_i : T_i$, $\text{AT} = T'_1 \times \dots \times T'_n$, $\Gamma \vdash T_i <_{\text{wdn}} T'_i$ for $i \in \{2, \dots, n\}$, and $\text{FirstFit}(\Gamma, m, T_1, T'_2 \times \dots \times T'_n) = \{(T'', MT'')\}$, and $MT'' = T_1 \times \dots \times T_n \rightarrow T$.
 - (a) e_1 is not ground. By induction hypothesis $< e_1, \sigma > \rightsquigarrow_p < e'_1, \sigma' >$.
 - i. e'_1 contains a `null` access: then the argument goes as in case 1.(a).
 - ii. there exists a Γ'', T''_1 , with Γ'' conforms to Γ , σ' conforms to Γ'' , $\Gamma'' \vdash e'_1 : T''_1$ and $\Gamma \vdash T''_1 <_{\text{wdn}} T_1$. The lemma can now be proven using $\Gamma' = \Gamma''$, and applying lemma 3.3.
 - (b) e_i are ground, for $i \in \{1, \dots, k-1\}$, and e_k is not ground. From induction hypothesis: $< e_k, \sigma > \rightsquigarrow_p < e'_k, \sigma' >$, and there exists a Γ' conforming to Γ , σ' conforms to Γ' , and there exists a T''_k with $\Gamma \vdash T''_k <_{\text{wdn}} T_k$. From the method call type rule in 4.5 it follows that $\Gamma, \sigma \vdash e_1.[\text{AT}]m(e_2 \dots e'_k \dots e_n) : T$, and the case is completed.
 - (c) e_i are ground, for $i \in \{1, \dots, n\}$. Then, there exists a C , with $\sigma(e_1) = \langle \langle \dots \rangle \rangle^C$, and $T_1 = C$. Because $\text{FirstFit}(\Gamma, m, C, T'_2 \times \dots \times T'_n) = \{(T'', MT'')\}$, by definition, $(T'', MT'') \in \text{MethDecls}(\Gamma, C, m)$. Therefore, by lemma 3.1, $\Gamma \vdash C \sqsubseteq T''$ and there exists T'_{n+1} with $\Gamma \vdash T'_{n+1} <_{\text{wdn}} T$, and $\text{MethBody}(\Gamma, p, C, m) = (T'', \lambda x_2 : T'_2, \dots, \lambda x_n : T'_n. \{ \text{stmts} \})$, and $\Gamma, \text{this} : T'', x_2 : T'_2, \dots, x_n : T'_n \vdash \text{stmts} : T'_{n+1}$. Now, choose z_1, \dots, z_n new in σ . We define $\Gamma' = \Gamma, z_1 : T'', z_2 : T'_2, \dots, z_n : T'_n$, and $\sigma' = \sigma[z_1 \mapsto e_1] \dots [z_n \mapsto e_n]$. Because z_i are new in σ , they are also new in Γ' . Therefore, Γ' conforms to Γ . Furthermore, because $\Gamma, \sigma \vdash e_1 : C$, $\Gamma \vdash C \sqsubseteq T''$, and because $\Gamma, \sigma \vdash e_i : T_i$, $\Gamma \vdash T_i <_{\text{wdn}} T'_i$, the new state σ' conforms to Γ' . Finally, a simple renaming exercise, in the assertion $\Gamma, \text{this} : T'', x_2 : T'_2, \dots, x_n : T'_n \vdash \text{stmts} : T'_{n+1}$ results in $\Gamma, z_1 : T'', z_2 : T'_2, \dots, z_n : T'_n \vdash \text{stmts}[z_1/\text{this}, z_2/x_2, \dots, z_n/x_n] : T'_{n+1}$. This completes the proof of this case.

Theorem 2 Soundness

Take any *Java_s* term τ , a well formed environment Γ , a type T with $\Gamma \vdash \tau : T$ a *Java_s* program p with $\Gamma \vdash p : \Gamma$, and a state σ that conforms to Γ .

There exists a *Java_{se}* program p' , $p' = \text{Comp}\{p, \Gamma\}$, a *Java_{se}* term τ' , and a state σ' , such that:

- $\langle \text{Comp}\{\tau, \Gamma\}, \sigma \rangle \rightsquigarrow_{\mathbf{p}'^*} \langle \tau', \sigma' \rangle$ and τ' contains a null access
or
- $T \neq \text{void}$, and $\langle \text{Comp}\{\tau, \Gamma\}, \sigma \rangle \rightsquigarrow_{\mathbf{p}'^*} \langle \tau', \sigma' \rangle$ and τ' is ground,
and $\exists T' : \Gamma, \sigma' \vdash \tau' : T', \quad \Gamma \vdash T' \langle_{\text{wdn}} T$ and σ' conforms to Γ
- $T = \text{void}$, and $\langle \text{Comp}\{\tau, \Gamma\}, \sigma \rangle \rightsquigarrow_{\mathbf{p}'^*} \langle \sigma' \rangle$ and σ' conforms to Γ

Proof follows from theorem 1, and lemmas 4.1, and 3.4.

6 An example

6.1 A Java program

Consider the following Java program:

```

class C1 {
    int x ;
    C2 f(C2 y){ x = 3; ... }
    void f(C3 y){ ... }
}
class C2 extends C1 {
    char u ;
    C3 f(C2 y){ x = 3; u = 'c'; ... }
}
class C3 extends C2 {
    C2 x ;
    C3 f(C2 y){ x = this; y = null ; ... }
}
C2 z ;

```

6.2 The corresponding Java_s environment and Java_s program

The corresponding Java_s environment Γ_0 is:

```

 $\Gamma_0 = \text{C1 ext Object } \{ x : \text{int}, f : \text{C2} \rightarrow \text{C2}, f : \text{C3} \rightarrow \text{void}, \}$ 
 $\text{C2 ext C1 } \{ u : \text{char}, f : \text{C2} \rightarrow \text{C3} \}$ 
 $\text{C3 ext C2 } \{ x : \text{C2}, f : \text{C2} \rightarrow \text{C3} \}$ 
 $z : \text{C2}$ 

```

The corresponding Java_s program is p:

```

p = {
    C1 ext Object { f is  $\lambda y:\text{C2}.\{ \text{this.x} := 3, \dots \}$ , f is  $\lambda y:\text{C3}.\{ \dots \}$ ,
    C2 ext C1 { f is  $\lambda y:\text{C2}.\{ \text{this.x} := 'e' ; \text{this.x} := 'c' \dots \}$ ,
    C3 ext C2 { f is  $\lambda y:\text{C2}.\{ \text{this.x} := \text{this}; y := \text{null} \dots \} \}$  }

```

6.3 The functions $\text{MethSigs}(,,)$, $\text{MethDecls}(,,)$, $\text{FieldDecl}(,,)$

For the environment Γ_0 , we obtain the following functions

$$\begin{aligned}
FieldDecl(\Gamma_0, C1, x) &= (C1, int) \\
FieldDecl(\Gamma_0, C2, x) &= (C1, int) \\
FieldDecl(\Gamma_0, C2, u) &= (C2, char) \\
FieldDecl(\Gamma_0, C3, x) &= (C3, C2) \\
FieldDecl(\Gamma_0, C3, u) &= (C2, char) \\
\\
MethDecls(\Gamma_0, C1, f) &= \{ (C1, C2 \rightarrow C2), (C1, C3 \rightarrow void) \} \\
MethDecls(\Gamma_0, C2, f) &= \{ (C2, C2 \rightarrow C3), (C1, C3 \rightarrow void) \} \\
MethDecls(\Gamma_0, C3, f) &= \{ (C3, C2 \rightarrow C3), (C1, C3 \rightarrow void) \} \\
\\
MethSigs(\Gamma_0, C1, f) &= \{ C2 \rightarrow C2, C3 \rightarrow void \} \\
MethSigs(\Gamma_0, C1, f) &= \{ C2 \rightarrow C3, C3 \rightarrow void \} \\
MethSigs(\Gamma_0, C1, f) &= \{ C2 \rightarrow C3, C3 \rightarrow void \}
\end{aligned}$$

6.4 The corresponding Java_{se} program

The program p would be mapped to p' , the following Java_{se} program:

$$\begin{aligned}
p' = Comp\{p, \Gamma_0\} = \{ \\
\quad C1 \text{ ext Object } \{ f \text{ is } \lambda y:C2. \{ \text{this.[C1]x := 3, \dots \}, f \text{ is } \lambda y:C3. \{ \dots \} \} \\
\quad C2 \text{ ext C1 } \{ f \text{ is } \lambda y:C2. \{ \text{this.[C1]x := 5 ; this.[C2]u := 'c' \dots \} \} \\
\quad C3 \text{ ext C2 } \{ f \text{ is } \lambda y:C2. \{ \text{this.[C3]x := this; y := null ; \dots \} \} \}
\end{aligned}$$

Furthermore, the Java_s term $z.f(z)$ would be extended as follows:

$$Comp\{z.f(z), \Gamma_0\} = z.[C2]f(z)$$

6.5 A possible state

The following state σ_0 conforms to the environment Γ_0 :

$$\begin{aligned}
\sigma_0(z) &= \text{addr}(2) \\
\sigma_0(\text{addr}(2)) &= \langle\langle x \ C1: \ -3, u \ C2: \ 'b', x \ C3: \ \text{addr}(4) \ \rangle\rangle^{C3} \\
\sigma_0(\text{addr}(4)) &= \langle\langle x \ C1: \ 2, u \ C2: \ 'c' \ \rangle\rangle^{C2}
\end{aligned}$$

6.6 Execution of a Java_{se} term

Let us assume we want to execute the Java_{se} term $z.[C2]f(z)$:

$$\begin{aligned}
\langle z.[C2]f(z), \sigma_0 \rangle &\rightsquigarrow_{p'} \langle \text{addr}(2).[C2]f(z), \sigma_0 \rangle &&\rightsquigarrow_{p'} \\
\langle \text{addr}(2).[C2]f(\text{addr}(2)), \sigma_0 \rangle &\rightsquigarrow_{p'} \langle (w.[C2]w' := z1; y := null ; \dots), \sigma_1 \rangle &&\rightsquigarrow_{p'} \\
\langle (y := null \dots), \sigma_2 \rangle &\rightsquigarrow_{p'} \langle (\dots), \sigma_3 \rangle
\end{aligned}$$

where σ_1 is :

$$\begin{aligned}
\sigma_1(z) &= \text{addr}(2) \\
\sigma_1(w) &= \text{addr}(2) \\
\sigma_1(w') &= \text{addr}(2) \\
\sigma_1(\text{addr}(2)) &= \langle\langle x \ C1: \ -3, u \ C2: \ 'b', x \ C3: \ \text{addr}(4) \ \rangle\rangle^{C3} \\
\sigma_0(\text{addr}(4)) &= \langle\langle x \ C1: \ 2, u \ C2: \ 'c' \ \rangle\rangle^{C2}
\end{aligned}$$

and where σ_2 is :

$$\begin{aligned}
\sigma_2(z) &= \text{addr}(2) \\
\sigma_2(w) &= \text{addr}(2) \\
\sigma_2(w') &= \text{addr}(2) \\
\sigma_2(\text{addr}(2)) &= \langle\langle x \ C1: \ -3, u \ C2: \ 'b', x \ C3: \ \text{addr}(2) \ \rangle\rangle^{C3} \\
\sigma_2(\text{addr}(4)) &= \langle\langle x \ C1: \ 2, u \ C2: \ 'c' \ \rangle\rangle^{C2}
\end{aligned}$$

and where σ_3 is :

$$\begin{aligned}\sigma_3(\mathbf{z}) &= \mathbf{addr}(2) \\ \sigma_3(\mathbf{w}) &= \mathbf{addr}(2) \\ \sigma_3(\mathbf{w}') &= \mathbf{null} \\ \sigma_3(\mathbf{addr}(2)) &= \langle\langle \mathbf{x} \text{ C1: } -3, \mathbf{u} \text{ C2: } 'b', \mathbf{x} \text{ C3: } \mathbf{addr}(2) \rangle\rangle^{\text{C3}} \\ \sigma_3(\mathbf{addr}(4)) &= \langle\langle \mathbf{x} \text{ C1: } 2, \mathbf{u} \text{ C2: } 'c' \rangle\rangle^{\text{C2}}\end{aligned}$$

If we consider the “environment extension” as in theorem 1, then in the third step of the reductions, we would have the environment $\Gamma' = \Gamma_0, \mathbf{w} : \text{C2}, \mathbf{w}' : \text{C2}$. The states σ_1, σ_2 and σ_3 conform to Γ' .

7 Conclusions and future work

We have given a formal description of the type system and operational semantics and type system of 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 obtain 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, starting with arrays, whose associated run-time checks can be described in a similar way to the description of `null` access. We also hope that our approach may serve as the basis for other studies on the language and possible extensions.

As an interesting subsidiary result, we formulated and prove important properties of compile time correct Java programs: Any class, which according to the type rules is widening another interface or class, must provide an implementation for any method declared in that superinterface or superclass. Even though the substitution property does not hold for Java expressions, it holds for expressions which have been enriched with compile time information. These properties are not stated as such in [14], but we believe that they were in the language designers' minds.

8 Acknowledgments

We are indebted to Peter Sellinger who gave us ample feedback, useful suggestions, and pointed out two major weaknesses in an earlier version of this paper. We are grateful to Guiseppe Castagna for feedback, and for pointing out alternative approaches, to an anonymous FOOL4 referee for feedback, to Steve Vickers for comments, and to Bernie Cohen for awakening our interest in the application of formal methods to Java. We would also like to express our appreciation 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] Derek Andrews and W. Henhapl. Pascal. In D. Bjorner and C. B. Jones, editors, *Formal Specification and Software Development*, pages 175–251. Prentice Hall International, 1982.
- [3] Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized types and java. Technical Report MIT LCS TM-553, MIT, May 1996.

- [4] 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.
- [5] John Boyland and Giuseppe Castagna. Type-safe compilation of covariant specialization: A practical case. In *ECOOP'96 Proceedings*, July 1996.
- [6] P. Canning, William Cook, and William Olthoff. Interfaces for object-oriented programming. In *OOPSLA '89*, pages 457–467, 1989.
- [7] Giuseppe Castagna. private communication, October 1996.
- [8] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 15 February 1995.
- [9] William Cook. A Proposal for making Eiffel Type-safe. In S. Cook, editor, *ECOOP'87 Proceedings*, pages 57–70. Cambridge University Press, July 1989.
- [10] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *POPL'90 Proceedings*, January 1990.
- [11] Luis Damas and Robin Milner. Principal Type Schemes for Functional Languages. In *POPL'82 Proceedings*, 1982.
- [12] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From hotjava to netscape and beyond. In *Security and Privacy'96 Proceedings*, May 1996.
- [13] Kathleen Fischer and John Mitchell. Class = Object and Datate Abstraction. Technical report, 1995.
- [14] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996.
- [15] R. Harper. A simplified account of polymorphic references. Technical Report CMU-CS-93-169, Carnegie Mellon University, 1993.
- [16] Daniel Ingalls. The smalltalk-76 programming system design and implementation. In *POPL'78 Proceedings*, pages 9–15, January 1978.
- [17] The Java home page, 1996.
- [18] The java language specification, October 1995.
- [19] The java language specification, May 1996.
- [20] Bertrand Meyer. Static typing and other mysteries of life, December 1995.
- [21] Modula-2 (base language), 1996.
- [22] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *POPL'97 Proceedings*, January 1997.
- [23] Peter Sellinger. private communication, October 1996.
- [24] Mads Tofte. Type Inference for Polymorphic References. In *Information and Computation'80 Conference Proceedings*, pages 1–34, November 1980.
- [25] A. van Wijngaarden et al. *Revised Report on the Algorithmic language ALGOL 68*. Springer-Verlag, 1976.