Towards an Operational Semantics and Proof of Type Soundness for Java

Sophia Drossopoulou and Susan Eisenbach
Department of Computing
Imperial College of Science, Technology and Medicine

1 Introduction

The philosophy of the Java language designers was to provide a small and simple language including only features with well known semantics. Java combines features from C++, Smalltalk, CLOS and other object oriented languages.

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, 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 [14] 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, 30, 32], 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, 22]. Given the growing usage of Java, it seems important that if there are loopholes in the type system they be discovered early on.

We 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 description in [16] which answered unambiguously all questions relating to the language subset we considered. 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 chapter we consider the following parts of the Java language: primitive types, classes and inheritance, instance variables and instance methods, inter-

faces, shadowing of instance variables, dynamic method binding, object creation with `new`, the `null` value, arrays, exceptions and exception handling.

We chose this 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 [5], the Damas and Milner polymorphic type systems for functional languages [10], and for the imperative extension [29]). We considered arrays, because of the known requirement for run-time type checking.

In contrast with our previous work [11, 12, 13], we follow the language description in [16] rather than the more general approach outlined in older versions of the language description.

1.2 Our approach

We define Java$_s$, a provably *safe* subset of Java containing the features listed previously, a term rewrite system to describe 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.

$$\begin{array}{ccccccc} \text{Java} & \supset \text{Java}_s & \xrightarrow[\mathcal{C}]{} & \text{Java}_{se} \subset \text{Java}_r & \rightsquigarrow_{\mathtt{p}} & \text{Java}_r \\ & \downarrow & & \downarrow \quad \downarrow & & \downarrow \\ & \text{Type} & = & \text{Type} \quad = \text{Type} & \geq_{wdn} & \text{Type} \end{array}$$

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, but do not diminish the applicability to Java itself. It only requires a straightforward transformation to turn a Java program from the domain under consideration to the corresponding Java$_s$ program.

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.* section 5.2. 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 chapter 15.11 in [16]). 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. Java$_{se}$ corresponds to Java$_A$ from chapter 4, and to Java$_{light}$ from chapter 5.

During execution, these terms may be rewritten to terms which are not expressible in Java$_{se}$. We therefore extend Java$_{se}$, obtaining Java$_r$, which describes *run-time* terms. Java$_r$ corresponds to Java$_R$ from chapter 4. In previous work [11, 12, 13] we did not distinguish between Java$_{se}$ and Java$_r$; instead, we

only considered one language with both the enrichments and the extensions. However, as Don Syme has pointed out, the two different reasons for language modifications should naturally lead to distinct languages. Also, such a distinction allows a clearer description of the concepts. Last but not least, this distinction is necessary for the formalization of the notions around binary compatibility [31].

The operational semantics is defined as a ternary rewrite relationship between configurations, programs and configurations. Configurations are tuples of Java_r 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 [18]), thus we do not need store types [1], or location typings [17]. 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.

There are strong links between our work and that described in chapters 4 and 5. Don Syme describes in chapter 4 the formalization of a large part of this work using his theorem checker, DECLARE. During this process he uncovered flaws in our work, which will be described later on. A close collaboration ensued.

David von Oheimb and Tobias Nipkow have encoded their formalization of an enriched language similar to Java_{se} into the theorem prover Isabelle. Thus the treatment of the original language, Java_s, is omitted. Their description of most language constructs is similar to ours, except for exceptions, for which they use a dedicated component of the run-time configuration. More importantly, they used a large-step operational semantics. This turned out to have incisive influence on the necessary proofs, and to allow for spectacular simplifications. Thus, in the large step semantics inconsistent intermediate states need not be considered and most lemmas could be significantly simplified. This difference came as a surprise to all authors. On the other hand, strictly speaking, large step semantics cannot make any promise about non-terminating programs not breaking the type system, nor is it yet clear how large step semantics could adequately describe co-routines.

The rest of this chapter is organized as follows: In section 2 we give an example in Java, which we use to illustrate the concepts introduced in the subsequent sections. In section 3 we give the syntax of Java_s. In section 4 we define the language Java_{se}. In section 5 we define the static types for Java_s, and the mapping from Java_s to Java_{se}. In section 6 we describe the types of Java_{se} terms. In section 7 we describe the extended language Java_r. In section 8 we describe states, configurations and the operational semantics for Java_r. In section 9 we give types to Java_r expressions. In section 10 we state properties of the operational semantics and in particular the Subject Reduction Theorem. In section 11 we draw some conclusions.

2 An example in Java

The following, admittedly contrived, Java program serves to demonstrate some of the Java features that we tackle, and will be used in later sections to illustrate our approach. It can have the following interpretation: Philosophers like truths. 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; they too may think together with another philosopher, and finally refer the question to another philosopher.

Assuming previous definitions of classes `Book`, `Food` and `Truth`, consider the classes `Phil`, `FrPhil` defined as:

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

Consider the following declarations and expressions:

```
     Phil aPhil ; FrPhil pascal = new FrPhil();
     ...aPhil.like
     ...aPhil.think(pascal)...aPhil.think(aPhil)
     ...pascal.like
     ...pascal.think(pascal)...pascal.think(aPhil)
```

The above example demonstrates:

- recursive scopes, *e.g.* the class `FrPhil` is visible inside the class `Phil`, that comes before its declaration;
- shadowing of instance variables by static types, *e.g.* `pascal.like` is an object of class `Food`, whereas `aPhil.like` indicates an object of class `Truth`, even after the assignment `aPhil:=pascal`;
- method binding according to the dynamic class of the receiver, and the static class of the arguments: The call `aPhil.think(pascal)` will result in calling the method `Phil::think(FrPhil)` (*i.e.* the `think` method declared in class `Phil` and which takes a `FrPhil` argument, and returns a `Book`), even if `aPhil` contains a pointer to a `FrPhil` object. The call `aPhil.think(aPhil)` will result in calling the method `Phil::think(Phil)` if `aPhil` is an object of class `Phil`, and it will result in calling `FrPhil::think(Phil)`, if `aPhil` is an object of class `FrPhil`. The call `pascal.think(pascal)` is ambiguous, because the methods `Phil::think(FrPhil)` and `FrPhil::think(Phil)` are applicable, and neither is more special than the other.

3 The language Java$_s$

Java$_s$ is a subset of Java, which includes 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 `NullPE`, arrays, array creation and the exceptions `ArrStoreE`, `NegSzeE` and `IndOutBndE`. The features we have not yet considered include initializers, constructors, finalizers, class variables and class methods, local variables, class modifiers, final/abstract classes and methods, `super`, strings, numeric promotions and widenings, concurrency, 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 (*i.e.* variable declarations, parameter and result types for methods, interfaces of classes) and evaluation information (*i.e.* 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 in the program (usually represented by a `p`).

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`) are entities that have been previously declared.

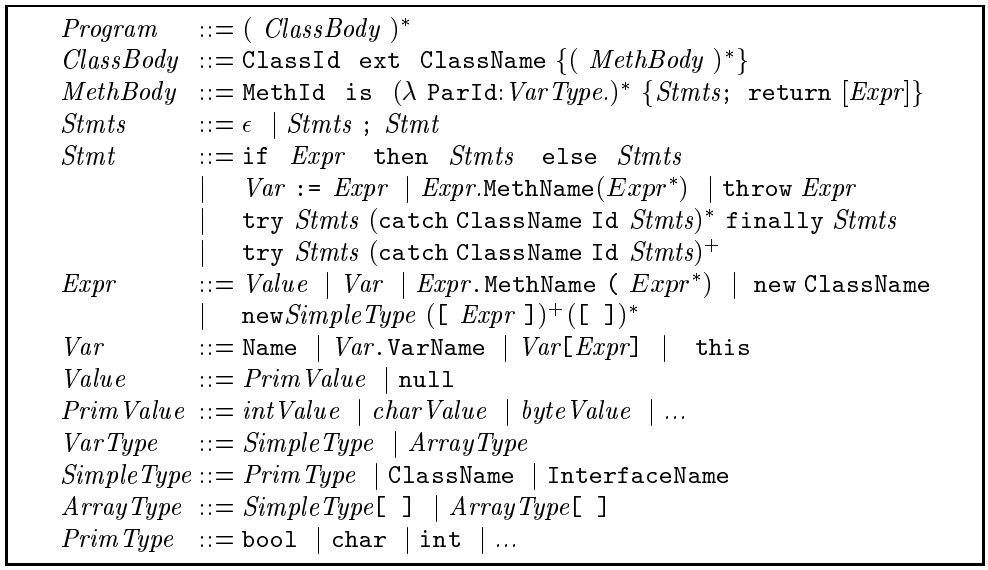
3.1 Java$_s$ Programs

A program, as described in figure 1, 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 only consider conditional statements, assignments, method calls, `try` and `throw` statements. This is because `loop`, `break`, `continue` and `case` statements can be coded in terms of conditionals and recursion.

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 $\texttt{new T}[\texttt{e}_1]\ldots[\texttt{e}_\texttt{n}][]_1\ldots[]_k$, $\texttt{n} \geq 1, k \geq 0$ 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 3.

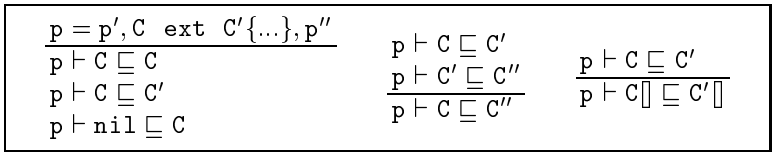
Java$_s$ programs contain the class hierarchy. Thus, from a program `p` we can deduce the $\sqsubseteq$ relationship, which is the transitive closure of the imme-



Program	::=	(*ClassBody*)*
ClassBody	::=	`ClassId ext ClassName` {(*MethBody*)*}
MethBody	::=	`MethId is` (λ `ParId`:*VarType*.)* {*Stmts*; `return` [*Expr*]}
Stmts	::=	ϵ | *Stmts* ; *Stmt*
Stmt	::=	`if` *Expr* `then` *Stmts* `else` *Stmts*
		| *Var* `:=` *Expr* | *Expr*.`MethName`(*Expr**) | `throw` *Expr*
		| `try` *Stmts* (`catch ClassName Id` *Stmts*)* `finally` *Stmts*
		| `try` *Stmts* (`catch ClassName Id` *Stmts*)$^+$
Expr	::=	*Value* | *Var* | *Expr*.`MethName (` *Expr**) | `new ClassName`
		| `new`*SimpleType* (`[` *Expr* `]`)$^+$(`[ ]`)*
Var	::=	`Name` | *Var*.`VarName` | *Var*`[`*Expr*`]` | `this`
Value	::=	*PrimValue* | `null`
PrimValue	::=	*intValue* | *charValue* | *byteValue* | ...
VarType	::=	*SimpleType* | *ArrayType*
SimpleType	::=	*PrimType* | `ClassName` | `InterfaceName`
ArrayType	::=	*SimpleType*`[ ]` | *ArrayType*`[ ]`
PrimType	::=	`bool` | `char` | `int` | ...

Fig. 1. Java$_s$ programs

diate superclass relation, and also applies to arrays whose component types are subclasses of each other. This relation is defined in figure 2. The notation $\mathtt{p} = \mathtt{p}', \mathtt{C\ ext\ C}'\{...\}, \mathtt{p}''$ means that p contains a declaration of C as a subclass of C′. The assertion $\mathtt{p} \vdash \mathtt{C} \sqsubseteq \mathtt{C}'$ indicates that for given program p, C is a subclass of C′. The functions p(C), which looks up a class body with identifier C in p, and



$$\frac{\mathtt{p} = \mathtt{p}', \mathtt{C\ ext\ C}'\{...\}, \mathtt{p}''}{\begin{array}{l}\mathtt{p} \vdash \mathtt{C} \sqsubseteq \mathtt{C} \\ \mathtt{p} \vdash \mathtt{C} \sqsubseteq \mathtt{C}' \\ \mathtt{p} \vdash \mathtt{nil} \sqsubseteq \mathtt{C}\end{array}} \qquad \frac{\mathtt{p} \vdash \mathtt{C} \sqsubseteq \mathtt{C}' \quad \mathtt{p} \vdash \mathtt{C}' \sqsubseteq \mathtt{C}''}{\mathtt{p} \vdash \mathtt{C} \sqsubseteq \mathtt{C}''} \qquad \frac{\mathtt{p} \vdash \mathtt{C} \sqsubseteq \mathtt{C}'}{\mathtt{p} \vdash \mathtt{C}[] \sqsubseteq \mathtt{C}'[]}$$

Fig. 2. subclasses deduced from programs p

Classes(p), which is the set of the identifiers of all classes defined in p.

Definition 1 *For a program* p*, we define* p(C)*, and Classes*(p) *as follows:*

- $\mathtt{p}(\mathtt{C}) = \mathtt{cBody}$ *iff* $\mathtt{p} = \mathtt{p}', \mathtt{cBody}, \mathtt{p}''$, *and* $\mathtt{cBody} = \mathtt{C\ ext\ C}'\ \mathtt{impl}\ ...\mathtt{I_n}\{...\}$,
- $\mathtt{p}(\mathtt{C}) = \mathtt{Undef}$, *otherwise.*
- $\mathtt{C} \in \mathit{Classes}(\Gamma)$ *iff* $p \vdash \mathtt{C} \sqsubseteq \mathtt{C}$.

3.2 Environments

The environment, defined in figure 3, 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, and all the classes described in chapters 20-22 of [16], *e.g.* the exception classes `Exception`, `NullPE`, `ArrStoreE`, `IndOutBndE`, `NegSzeE` and others – we do not need to distinguish between checked and unchecked exceptions. Declarations may be class declarations, interface declarations or identifier declarations.

Env	::=	*StandardEnv* | *Env* ; *Decl*
StandardEnv	::=	`Exception ext Object` ...`NullPE ext Exception` ...; ...
Decl	::=	`ClassId ext ClassName impl` (`InterfName`)*
		{(`VarId` : *VarType*)* (`MethId` : *MethType*)*}
	|	`InterfId ext InterfName`*{(`MethId` : *MethType*)*}
	|	`VarId` : *VarType*
MethType	::=	*ArgType* $\rightarrow$ (*VarType* | `void`)
ArgType	::=	[*VarType* ($\times$*VarType*)*]
VarType	::=	*SimpleType* | *ArrayType*
SimpleType	::=	*PrimType* | `ClassName` | `InterfaceName`
ArrayType	::=	*SimpleType*`[ ]` | *ArrayType*`[ ]`
PrimType	::=	`bool` | `char` | `int` | ...
Type	::=	*VarType* | `void` | `nil` | *MethType* | `ClassName-Thrn`

Fig. 3. Environments

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. Since 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 have not been considered. Variable declarations introduce variables of a given type.

3.3 The example in Java$_s$

The Java philosophers classes from section 2 correspond to the following Java$_s$ program $\mathtt{p}_s$, and environment Γ_0:

```
p_s = Phil  ext  Object {
        think is λy:Phil.{...}
        think is λy:FrPhil.{...}
    }
    FrPhil ext Phil {
        think is λy:Phil.{this.like :=oyster;...}
    }

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

3.4 Subclasses, subinterfaces, widening

The subclass $\sqsubseteq$ and the implements $:_{imp}$ relations deduced from an environment Γ are defined by the inference rules in figure 4.

$$\frac{\Gamma = \Gamma',\ \mathtt{C}\ \ \mathtt{ext}\ \ \mathtt{C}'\ \ \mathtt{impl}\ ...\mathtt{I}...\{...\}, \Gamma''}{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C},\quad \Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}',\quad \Gamma \vdash \mathtt{C} :_{imp} \mathtt{I}}$$

$$\frac{\Gamma = \Gamma',\ \mathtt{I}\ \ \mathtt{ext}\ ...,\ \mathtt{I}'\ , ...\{...\}, \Gamma''}{\Gamma \vdash \mathtt{I} \leq \mathtt{I},\quad \Gamma \vdash \mathtt{I} \leq \mathtt{I}'}$$

$$\frac{}{\vdash \mathtt{Object} \sqsubseteq \mathtt{Object}} \qquad \frac{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}' \quad \Gamma \vdash \mathtt{C}' \sqsubseteq \mathtt{C}''}{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}''} \qquad \frac{\Gamma \vdash \mathtt{I} \leq \mathtt{I}' \quad \Gamma \vdash \mathtt{I}' \leq \mathtt{I}''}{\Gamma \vdash \mathtt{I} \leq \mathtt{I}''}$$

Fig. 4. subclasses and subinterfaces

By the assertion $\Gamma = \Gamma'$, *def*, Γ'' we indicate that Γ contains the definition *def*. Every class introduced in Γ is its own subclass, and the assertion $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}$ 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 $\Gamma \vdash \mathtt{C} :_{imp} \mathtt{I}$ 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 $\Gamma \vdash \mathtt{I} \leq \mathtt{I}$ 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.

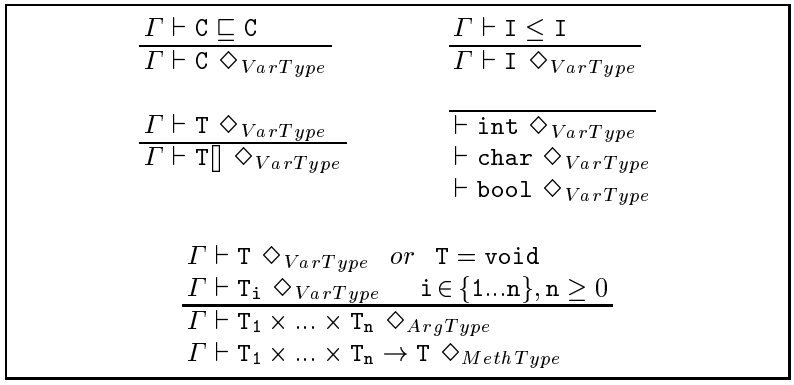


Fig. 5. variable and method types

Variable types, *i.e.* primitive types, interfaces, classes and arrays, as can be seen in figure 5, are required in type declarations. Method types, *i.e.* `n` argument types and a result type, also defined in figure 5, 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. Note that we do not keep track of potentially throwable exceptions in the method type. However, in future work method types should be extended to do so, and a stronger subject reduction theorem should be proven, stating that a checked exception can only be thrown during execution of a method that mentions this exception's class (or superclass) in the method's type.

The widening relationship, described in figure 6, exists between variable types. If a type `T` can be widened to a type `T'` (expressed as $\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T'}$), 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 [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 `nil` which can be widened to any array, class or interface.

3.5 Well-formed declarations and environments

The relations $\sqsubseteq$, $:_{imp}$, $\leq$ and $\leq_{wdn}$ are computable for any environment – as can be shown straightforwardly. In figure 7 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 the philosophers example, class `Phil` uses the class `FrPhil` whose declaration follows that of `Phil`). We shall call Γ well-formed, iff $\Gamma \vdash \Gamma \diamond$, in which case we use the shorthand $\Gamma \vdash \diamond$, *c.f.* the third rule in figure 7. The assertion $\Gamma \vdash \Gamma' \diamond$ is

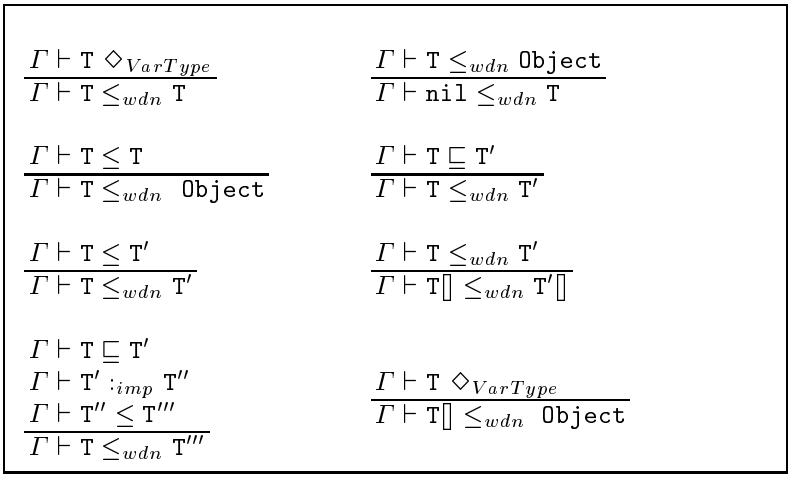


Fig. 6. the widening relationship

checked in two stages: The first stage establishes the relations $\sqsubseteq$, $:_{imp}$, $\leq$ and $\leq_{wdn}$ for the complete environment Γ and establishes that $\sqsubseteq$ and $\leq$ are acyclic; if this is the case, then 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. This is expressed by the first rule in figure 7.

We need the notion of definition table lookup, *i.e.* $\Gamma(\mathtt{Id})$, which returns the definition of the identifier $\mathtt{Id}$ in Γ, if it has one.

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

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

Furthermore, $Vars(\Gamma)$, $Classes(\Gamma)$, and $Interfaces(\Gamma)$ contain the identifiers of all variables, classes or interfaces declared in Γ, i.e.

- $\mathtt{x} \in Vars(\Gamma)$ *iff* $\Gamma = \Gamma', \mathtt{x} : \mathtt{T}, \Gamma''$.
- $\mathtt{C} \in Classes(\Gamma)$ *iff* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}$.
- $\mathtt{I} \in Interfaces(\Gamma)$ *iff* $\Gamma \vdash \mathtt{I} \leq \mathtt{I}$.

A variable should be declared to have a variable type and it should be declared only once, *c.f.* the second rule in figure 7. The type declaration for `T` may follow textually that of the variable `x`, as for example in:

`A x; ...class A ....`

We now consider when class declarations are well-formed. For this we shall need several auxiliary concepts. The following auxiliary definition allows the extraction of the argument types and the result type from a method type and helps us describe restrictions imposed on variable and method definitions for classes or interfaces, given in chapters 8.2 and 9 in [16].

Definition 3 *For a method type* $\mathtt{MT} = \mathtt{T}_1 \times ... \times \mathtt{T}_n \to \mathtt{T}$*, we define the* argument types *and the* result type*:*

- $\mathit{Args}(\mathtt{MT}) = \mathtt{T}_1 \times ... \times \mathtt{T}_n$
- $\mathit{Res}(\mathtt{MT}) = \mathtt{T}$

Next we introduce some functions to find the class components:

- $\mathit{FDec}(\Gamma, \mathtt{C}, \mathtt{v})$ indicates the nearest superclass of `C` (possibly `C` itself) which contains a declaration of the instance variable `v` and its declared type;
- $\mathit{FDecs}(\Gamma, \mathtt{C}, \mathtt{v})$ indicates all the field declarations for `v`, which were declared in a superclass of `C`, and possibly hidden by `C`, or another superclass.
- $\mathit{MDecs}(\Gamma, \mathtt{C}, \mathtt{m})$ 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;
- $\mathit{MSigs}(\Gamma, \mathtt{C}, \mathtt{m})$ returns all signatures for method `m` in class `C`, or inherited and not hidden by any of its superclasses.

Note that shadowed variables are treated differently from overridden methods. Namely, shadowed variables are part of the set *FDecs*, whereas overridden methods are not part of the set *MDecs*. The reason for the difference is that shadowed variables need to be stored in the objects of subclasses (*e.g.* a `FrPhil` object contains a `like` field inherited from the class `Phil`, even though this field is shadowed in `FrPhil`), whereas overridden methods are never called by objects of the subclasses (*e.g.* for `FrPhil` objects the only `think` method with a `Phil` argument is that from `FrPhil`, whereas that defined in `Phil` is of no interest to `FrPhil` objects).

From now on, we implicitly expect Γ to have unique declarations and the relations $\sqsubseteq$ and $\leq$ to be acyclic up to reflexivity. Thus the functions *FDec*, *FDecs*, *MDecs* and *MSigs* are well-defined, *c.f.* [26].

Definition 4 *For an environment* Γ*, with a class declaration for* `C`*, i.e.* $\Gamma = \Gamma', \mathtt{C}\ \mathtt{ext}\ \mathtt{C}'\ \mathtt{impl}\ \mathtt{I}_1, ...\mathtt{I}_n\{\mathtt{v}_1 : \mathtt{T}_1, ...\mathtt{v}_k : \mathtt{T}_k, \mathtt{m}_1 : \mathtt{MT}_1, ...\mathtt{m}_l : \mathtt{MT}_l\}, \Gamma''$*, define:*

– $FDec(\Gamma, \texttt{Object}, \texttt{v}) = \texttt{Undef}$ *for any* $\texttt{v}$
$FDec(\Gamma, \texttt{C}, \texttt{v}) = (\texttt{C}, \texttt{T}_\texttt{j}) \quad \textit{iff} \quad \texttt{v} = \texttt{v}_\texttt{j}$
$FDec(\Gamma, \texttt{C}, \texttt{v}) = FDec(\Gamma, \texttt{C}', \texttt{v}) \quad \textit{iff} \quad \texttt{v} \neq \texttt{v}_\texttt{j}\ \forall \texttt{j} \in \{\texttt{1...k}\}$

– $FDecs(\Gamma, \texttt{Object}, \texttt{v}) = \emptyset$
$FDecs(\Gamma, \texttt{C}, \texttt{v}) = \{(C', T') \mid (C', T') = FDec(\Gamma, \texttt{C}, \texttt{v})\} \cup FDecs(\Gamma, \texttt{C}', \texttt{v})$

– $MDecs(\Gamma, \texttt{Object}, \texttt{m}) = \emptyset$
$MDecs(\Gamma, \texttt{C}, \texttt{m}) = \{\ (\texttt{C}, \texttt{MT}_\texttt{j}) \quad | \quad \texttt{m} = \texttt{m}_\texttt{j}\ \} \cup$
$\{\ (\texttt{C}'', \texttt{MT}'') \quad | \quad (\texttt{C}'', \texttt{MT}'') \in MDecs(\Gamma, \texttt{C}', \texttt{m}), \quad \textit{and}$
$\forall \texttt{j} \in \{\texttt{1...l}\} : \texttt{m} = \texttt{m}_\texttt{j} \quad \Longrightarrow \quad Args(\texttt{MT}_\texttt{j}) \neq Args(\texttt{MT}'')\ \}$

– $MSigs(\Gamma, \texttt{C}, \texttt{m}) = \{\ \texttt{MT} \quad | \quad \exists \texttt{C}''\ \textit{with}\ (\texttt{C}'', \texttt{MT}) \in MDecs(\Gamma, \texttt{C}, \texttt{m})\ \}$

The sets $FDecs(\Gamma, \texttt{Object}, \texttt{v})$ and $MDecs(\Gamma, \texttt{Object}, \texttt{m})$ should contain the entities described in chapter 20.1 of [16]. We defined them as empty sets for simplicity.

For the philosophers example the above functions are:

$FDec(\Gamma_0, \texttt{Phil}, \texttt{like}) = \texttt{(Phil,Truth)}$
$FDec(\Gamma_0, \texttt{FrPhil}, \texttt{like}) = \texttt{(FrPhil,Food)}$
$FDecs(\Gamma_0, \texttt{Phil}, \texttt{like}) = \{\texttt{(Phil,Truth)}\}$
$FDecs(\Gamma_0, \texttt{FrPhil}, \texttt{like}) = \{\texttt{(Phil,Truth)}, \texttt{(FrPhil,Food)}\}$
$MDecs(\Gamma_0, \texttt{Phil}, \texttt{think}) = \{\texttt{(Phil,Phil}\rightarrow\texttt{Phil)}, \texttt{(Phil,FrPhil}\rightarrow\texttt{Book)}\}$
$MDecs(\Gamma_0, \texttt{FrPhil}, \texttt{think}) = \{\texttt{(FrPhil,Phil}\rightarrow\texttt{Phil)}, \texttt{(Phil,FrPhil}\rightarrow\texttt{Book)}\}$
$MSigs(\Gamma_0, \texttt{Phil}, \texttt{think}) = \{\texttt{Phil}\rightarrow\texttt{Phil}, \texttt{FrPhil}\rightarrow\texttt{Book}\}$

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

Definition 5 *For an environment Γ, containing an interface declaration for* $\texttt{I}$, *i.e.* $\Gamma = \Gamma', \texttt{I ext I}_1, ...\texttt{I}_\texttt{n}\{\texttt{m}_1 : \texttt{MT}_1, ...\texttt{m}_\texttt{k} : \texttt{MT}_\texttt{k}\}, \Gamma''$, *we define:*

– $MDecs(\Gamma, \texttt{I}, \texttt{m}) = \{\ (\texttt{I}, \texttt{MT}_\texttt{j}) \quad | \quad \texttt{m} = \texttt{m}_\texttt{j}\ \} \cup$
$\{\ (\texttt{I}', \texttt{MT}') \quad | \quad \exists \texttt{j} \in \{\texttt{1...n}\} : \quad (\texttt{I}', \texttt{MT}') \in MDecs(\Gamma, \texttt{I}_\texttt{j}, \texttt{m})$
$\textit{and} \quad \forall \texttt{i} \in \{\texttt{1...k}\}\ \ \texttt{m} = \texttt{m}_\texttt{i} \quad \Longrightarrow \quad Args(\texttt{MT}') \neq Args(\texttt{MT}_\texttt{i})\ \}$

– $MSigs(\Gamma, \texttt{I}, \texttt{m}) = \{\ \texttt{MT} \quad | \quad \exists \texttt{I}' : \quad (\texttt{I}', \texttt{MT}) \in MDecs(\Gamma, \texttt{I}, \texttt{m})\ \}$

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

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

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

$$\frac{\begin{array}{l}\mathtt{n} \geq 0, \mathtt{l} \geq 0 \\ \Gamma \vdash \Gamma' \diamond \\ \Gamma'(\mathtt{I}) = \mathtt{Undef} \\ NOT\ \ \Gamma \vdash \mathtt{I_i} \leq \mathtt{I}\ \ \mathtt{j} \in \{1...\mathtt{n}\} \\ \Gamma \vdash \mathtt{I_j} \leq \mathtt{I_j}\ \ \mathtt{j} \in \{1...\mathtt{n}\} \\ \Gamma \vdash \mathtt{MT_j} \diamond_{MethType}\ \ \mathtt{j} \in \{1...\mathtt{l}\} \\ \mathtt{m_i} = \mathtt{m_j} \implies \mathtt{i} = \mathtt{j}\ \text{ or }\ Args(\mathtt{MT_i}) \neq Args(\mathtt{MT_j}) \\ \mathtt{i} \in \{1...\mathtt{n}\}, \mathtt{j} \in \{1...\mathtt{l}\}\ \ \mathtt{MT} \in MSigs(\Gamma, \mathtt{I_i}, \mathtt{m_j}),\ Args(\mathtt{MT}) = Args(\mathtt{MT_j}) \\ \qquad \implies Res(\mathtt{MT_j}) = Res(\mathtt{MT}) \\ \forall \mathtt{i}, \mathtt{j} \in \{1...\mathtt{n}\}\ \ \mathtt{MT_1} \in MSigs(\Gamma, \mathtt{I_1}, \mathtt{m}),\ \ \mathtt{MT_2} \in MSigs(\Gamma, \mathtt{I_2}, \mathtt{m}) : \\ \quad Args(\mathtt{MT_1}) : Args(\mathtt{MT_2}) \implies Res(\mathtt{MT_1}) = Res(\mathtt{MT_2})\end{array}}{\Gamma \vdash \Gamma', \mathtt{I\ ext\ I_1}, ...\mathtt{I_n}\{\ \mathtt{m_1} : \mathtt{MT_1}, ...\mathtt{m_l} : \mathtt{MT_l}\} \diamond}$$

Fig. 7. well-formed environments

reduction theorem when proving that there exists a redex in any well-typed non-ground term.

Lemma 1 *For any environment* Γ*, types* $\mathtt{T}$, $\mathtt{T}'$ *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* $\Gamma(\mathtt{T}') = \mathtt{T}'\ \ \mathtt{ext}\ ...\ \mathtt{impl}\ ...\{...\mathtt{m} : \mathtt{MT}...\}$ *and* $\forall \mathtt{T}'', \mathtt{C} \neq \mathtt{T}'$ *with* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{T}',\ \Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{C}$:

$\Gamma(\mathtt{C}) \neq \mathtt{C}\ \mathtt{ext}\ \ldots\ \mathtt{impl}\ \ldots\{\ldots \mathtt{m} : \mathit{Args}(\mathtt{MT}) \rightarrow \mathtt{T}''\}$

or

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

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

The language description [16] imposes the following requirements, when a new class C is declared as

$$\mathtt{C}\ \ \mathtt{ext}\ \ \mathtt{C}'\ \mathtt{impl}\ \ \mathtt{I}_1, \ldots \mathtt{I}_n\{\mathtt{v}_1 : \mathtt{T}_1, \ldots \mathtt{v}_k : \mathtt{T}_k, \mathtt{m}_1 : \mathtt{MT}_1, \ldots \mathtt{m}_l : \mathtt{MT}_l\}$$

– 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 C

– there are no cyclic subclass dependencies between $\mathtt{C}'$ and 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 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 the same result type as the overridden method;

– "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".

These requirements are formalized in the fourth rule in figure 7. Similar requirements for interfaces are given in [16], and their formalization is also given in the fifth rule in figure 7.

3.6 Properties of well-formed environments

It is straightforward to state and prove the following properties of well-formed environments: Two types that are in the subclass relationship are classes, $\sqsubseteq$ is reflexive, transitive and antisymmetric, and the subclass hierarchy forms a tree. Also, two types that are in the subinterface relationship are interfaces, and $\leq$ is transitive, reflexive and antisymmetric. Unlike $\sqsubseteq$, $\leq$ does not form a tree.

Widening is reflexive, transitive and antisymmetric. If an interface widens to another type, then the second type is a superinterface of the first. If a type widens to a class, then the type is a subclass of that class. If a class widens to an interface `I`, then the class implements a subinterface of `I`. 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.

Finally, the following lemma states that 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 and return type as the method from `T'`.

Lemma 2 *If* $\Gamma \vdash \diamond$, $\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'$, *and* $\mathtt{MT} \in \mathit{MSigs}(\Gamma, \mathtt{T}', \mathtt{m})$, *then:*

$$\mathtt{MT} \in \mathit{MSigs}(\Gamma, \mathtt{T}, \mathtt{m})$$

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

4 Java$_{se}$ – enriching Java$_s$

Java$_{se}$ is an enriched version of Java$_s$ which provides compile-time type information necessary at run-time. It corresponds to Java$_A$ from chapter 4, and to Java$_{light}$ from chapter 5. The syntax of Java$_{se}$ programs is given in figure 8.

The Java$_{se}$ syntax is identical to that of Java$_s$, except for enrichments required in the following four cases. Method calls are enriched by the signature of the most special applicable method available at compile-time. This is why, the Java$_s$ syntax *Expr*`.MethName(`*Expr**`)` is replaced in Java$_{se}$ by the syntax *Expr*`.[`*ArgType*`]MethName(`*Expr**`)`. Instance variable accesses are enriched by the class containing the field declaration. Thus, *Expr*`.VarName` is replaced in Java$_{se}$ by *Expr*`.[ClassName]VarName`. Object creation is enriched by the names of all its fields, the classes they were declared in and their initial, default values. Therefore, the Java$_s$ syntax `new ClassName` is replaced by the Java$_{se}$ syntax `new ClassName` $\ll$`(VarName ClassName Value)`$^*\gg$. Finally, array creation is enriched by the initial values to be stored in each component of the new array. Therefore, the Java$_s$ syntax `new` *SimpleType* `[` *Expr* `]`$^+$`[ ]`* is replaced in Java$_{se}$ by `new` *SimpleType* `[`*Expr*`]`$^+$`[]`$^*[\![$*Value*$]\!]$. Examples of enriching method calls, instance variable access and of object creation can be seen in section 4.1. The Java$_s$ array creation `new int[3]` is represented in Java$_{se}$ as `new int[3]`$[\![$`0`$]\!]$.

Program	::=	(*ClassBody*)*
ClassBody	::=	`ClassId ext ClassName` {(*MethBody*)*}
MethBody	::=	`MethId is` (λ `ParId` : *VarType.*)* {*Stmts* `;` `return` [*Expr*] }
Stmts	::=	ϵ | *Stmts* `;` *Stmt*
Stmt	::=	`if` *Expr* `then` *Stmts* `else` *Stmts*
	|	*Var* `:=` *Expr* | *Expr.*`MethName`(*Expr**) | `throw` *Expr*
	|	`try` *Stmts* (`catch ClassName Id` *Stmts*)* `finally` *Stmts*
	|	`try` *Stmts* (`catch ClassName Id` *Stmts*)$^+$
Type	::=	*VarType* | `void` | `nil` | `ClassName-Thrn`
Expr	::=	*Value* | *Var*
	|	*Expr.*[*ArgType*]`MethName`(*Expr**)
	|	`new ClassName` $\ll$(`VarName ClassName Value`)$^*\gg$
	|	`new` *SimpleType* ([*Expr*])$^+$([])$^*[\![$*Value*$]\!]$
Var	::=	`Name` | *Var*`[`*Expr*`]` | `this`
	|	*Var.*[`ClassName`]`VarName`
Value	::=	*PrimValue* | `null`
PrimValue	::=	*intValue* | *charValue* | *byteValue* | ...
VarType	::=	*SimpleType* | *ArrayType*
SimpleType	::=	*PrimType* | `ClassName` | `InterfaceName`
ArrayType	::=	*SimpleType*`[ ]` | *ArrayType*`[ ]`
PrimType	::=	`bool` | `char` | `int` | ...

Fig. 8. Java$_{se}$ programs

4.1 The example in Java$_{se}$

The program $\mathtt{p}_s$ from section 2 is mapped to the Java$_{se}$ program $\mathtt{p}_{se}$:

$\mathtt{p}_{se} = \mathcal{C}[\![\Gamma_0, \mathtt{p}_s]\!] =$

```
Phil ext Object{
        think is λy:Phil.{...}
        think is λy:FrPhil.{...}
}
FrPhil ext Phil {
        think is λy:Phil.{ this.[FrPhil]like :=oyster; ...}
}
```

The terms would be represented as:

```
...  pascal := new FrPhil ≪ like Phil nil, like FrPhil nil ≫
...  aPhil.[Phil]like ...
...  aPhil.[Phil]think (aPhil)
...  aPhil.[FrPhil]think (pascal) ...
...  pascal.[FrPhil]like ...
```

... `pascal.think(pascal) !!` ambiguous call
... `pascal.[Phil]think (aPhil)` ...

5 Java$_s$ types

The type rules for Java$_s$ are given in figures 9, 10, and 11. They correspond to the type checking phase of a Java compiler and have the form $\Gamma \vdash \mathtt{t} : \mathtt{T}$, which means that term `t` has the type `T` in the environment Γ. The assertion $\Gamma \vdash \mathtt{p} \diamond$ signifies that program `p` is well-formed under the environment Γ (*i.e.* that all expression are type-correct, and that all classes conform to their definitions), whereas $\Gamma \vdash \mathtt{p} \diamond\!\!\diamond$ signifies that `p` is complete, (*i.e.* well-formed, and it provides a class body for each class declared in Γ). In parallel with type checking, the program is enriched with type information, described by the mapping $\mathcal{C}$:

$$\mathcal{C} : \text{Java}_s \times \text{Environment} \longrightarrow \text{Java}_{se}$$

Thus, each type rule is followed by an enrichment equation of the form $\mathcal{C}\{\!\{\Gamma, \mathtt{t}\}\!\} = \mathtt{t}'$ meaning that the Java$_s$ term `t` is enriched to the equivalent Java$_{se}$ term `t`$'$. The enrichment rules are given together with the type rules because in some cases (*i.e.* for method call and field access) the enrichments use type information.

Figure 9 describes the types for variables, primitive values, `null`, statements, newly created objects and arrays, and field and array access. According to the first rule, character literals have type character, integer literals have type integer *etc.* According to the second rule, a statement sequence has the same type as its last statement. A return statement has type `void`, or the same type as the expression it returns. An expression of type `T`$'$ can be assigned to a variable of a type `T` if `T`$'$ can be widened to `T`. A conditional consists of two statement sequences not necessarily of the same type.

For a class `C`, the expression `new C` has type `C`. For a simple type `T`, the expression `new T`$[\mathtt{e}_1]\ldots[\mathtt{e}_n][]_1\ldots[]_k$ is a `n+k`-dimensional array of elements of type `T`. Array and object creation expressions are enriched with initialization information that determine the values for component initialization. Initial values are defined in ch. 4.5.5. of [16], and here in the following definition:

Definition 6 *The* initial value *of a simple type is*:

- `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 classes, interfaces or* `nil`

For an array access `v`$[\mathtt{e}]$, the variable `v` should have an array type `T`$[]$, and `e` should be of integer type. For a field access `v.f`, the variable `v` should have a class

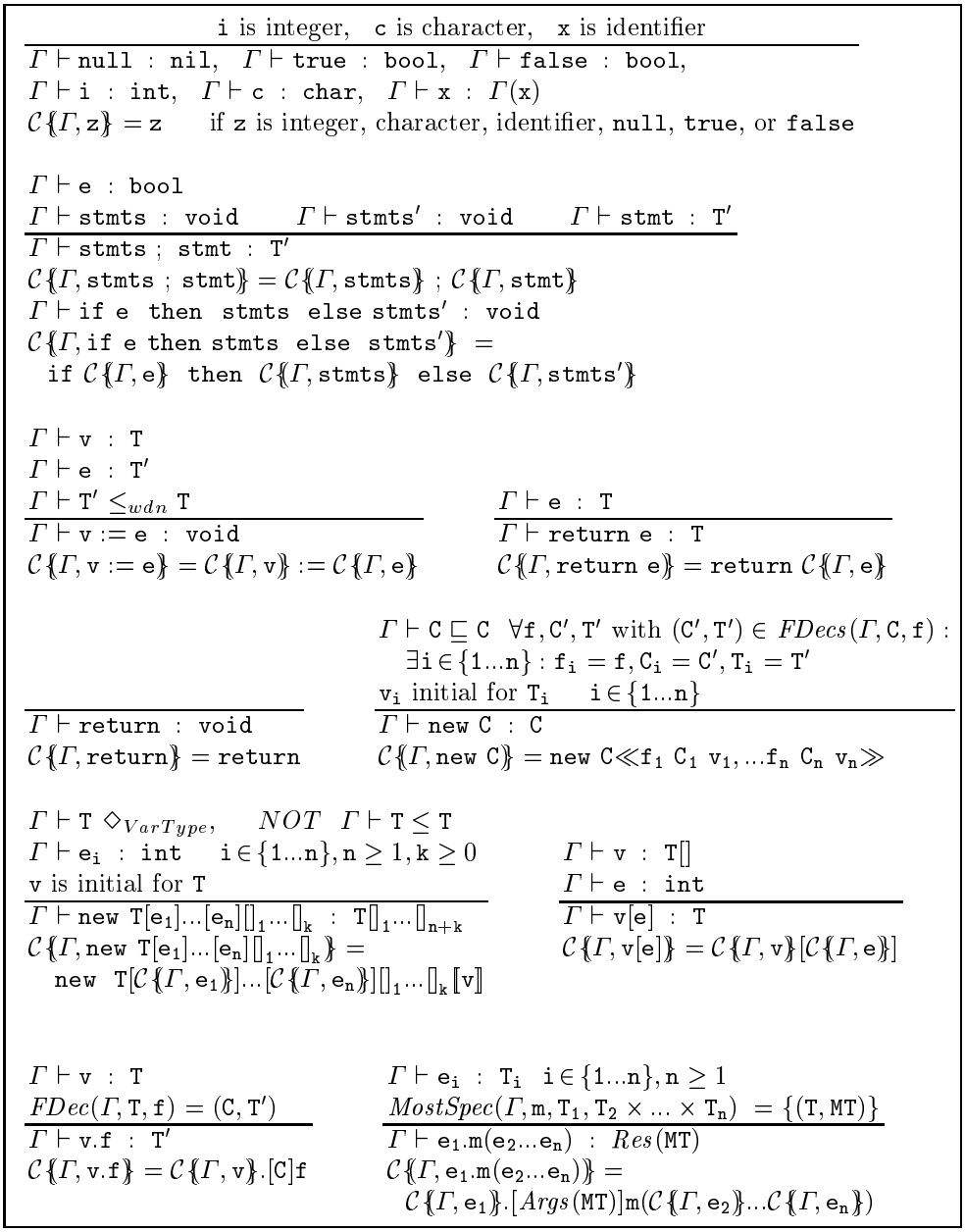


Fig. 9. types for Java$_s$ expressions and statements

type $\mathtt{T}$, (because we have only considered non-static fields, in Java$_s$ only instances have fields) one of whose superclasses ($\mathtt{C}$) should contain a field declaration for $\mathtt{f}$ of type $\mathtt{T}'$, *i.e.* $FDec(\Gamma, \mathtt{T}, \mathtt{f}) = (\mathtt{C}, \mathtt{T}')$, in which case the field access expression has type $\mathtt{T}'$, and the information from which superclass the field declaration is inherited is stored in the corresponding Java$_{se}$ expression, *i.e.* $\mathcal{C}\{\!\{\Gamma, \mathtt{v.f}\}\!\} = \mathcal{C}\{\!\{\Gamma, \mathtt{v}\}\!\}.[\mathtt{C}]\mathtt{f}$.

Figure 10 contains the type rules for method bodies and method calls, as in 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, if and only if it is defined in a subclass or subinterface and all argument types can be widened to 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 7 *For an environment Γ, identifier $\mathtt{m}$, variable types $\mathtt{T}$, $\mathtt{T}_1$, ... $\mathtt{T}_n$, the* most special declarations *are defined as follows:*

- $ApplMeths(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T}_1 \times ... \times \mathtt{T}_n) = \{(\mathtt{T}', \mathtt{MT}') \mid (\mathtt{T}', \mathtt{MT}') \in MDecs(\Gamma, \mathtt{T}, \mathtt{m})$ *and* $\mathtt{MT}' = \mathtt{T}'_1 \times ... \times \mathtt{T}'_n \to \mathtt{T}'_{n+1}$ *and* $\Gamma \vdash \mathtt{T}_i \leq_{wdn} \mathtt{T}'_i$ *for* $\mathtt{i} \in \{1...n\}\}$
- $(\mathtt{T}, \mathtt{T}_1 \times ... \times \mathtt{T}_n \to \mathtt{T}_{n+1})$ *is* more special than $(\mathtt{T}', \mathtt{T}'_1 \times ... \times \mathtt{T}'_n \to \mathtt{T}'_{n+1})$ *iff* $\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'$ *and* $\forall \mathtt{i} \in \{1...n\}\ \Gamma \vdash \mathtt{T}_i \leq_{wdn} \mathtt{T}'_i$
- $MostSpec(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T}_1 \times ... \times \mathtt{T}_n) = \{(\mathtt{T}', \mathtt{MT}') \mid (\mathtt{T}', \mathtt{MT}') \in ApplMeths(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T}_1 \times ... \times \mathtt{T}_n)$ *and* *if* $(\mathtt{T}'', \mathtt{MT}'') \in ApplMeths(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T}_1 \times ... \times \mathtt{T}_n)$ *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 if this set contains exactly one pair. The argument types of the signature of this pair is stored as the *method descriptor*, *c.f.* ch.15.11 in [16], and the result type of the signature is the type of the message expression.

Figure 10 describes the types for program, method and class bodies. The first rule describes the type of a method body with parameters $\mathtt{x}_1$, ..., $\mathtt{x}_n$, consisting of the statements $\mathtt{stmts}$. The renaming of variables in the method body, namely $\mathtt{stmts}[\mathtt{z}_1/\mathtt{x}_1, ..., \mathtt{z}_n/\mathtt{x}_n]$, is necessary in order to avoid name clashes and, also, in order for lemma 8 to hold – as pointed out in [25]. It is worth noticing that the rules describing method bodies do not determine $\mathtt{T}$ – instead, the expected return type of the method, $\mathtt{T}$, is taken from the environment Γ when applying the next rule of the figure, which describes class bodies.

The second rule in figure 10 describes the type of a class body consisting of method bodies $\mathtt{mBody}_1$, ... $\mathtt{mBody}_n$. Note that each $\mathtt{mBody}_i$ is type checked in the environment Γ, $\mathtt{this} : \mathtt{C}$, which does not contain the instance variable declarations $\mathtt{v}_1 : \mathtt{T}_1$..., $\mathtt{v}_k$: $\mathtt{T}_k$. Thus, through the type system, we force the use of the expression $\mathtt{this.v}_j$ as opposed to $\mathtt{v}_j$.

$$\frac{\begin{array}{l}\texttt{mBody} = \texttt{m is } \lambda \texttt{x}_1 : \texttt{T}_1 ... \lambda \texttt{x}_\texttt{n} : \texttt{T}_\texttt{n}.\{\texttt{stmts}\} \\ \texttt{x}_\texttt{i} \neq \texttt{this} \quad \texttt{i} \in \{1...\texttt{n}\} \\ \texttt{z}_1, ..., \texttt{z}_\texttt{n} \text{ are new variables in } \Gamma \\ \texttt{stmts}' = \texttt{stmts}[\texttt{z}_1/\texttt{x}_1, ..., \texttt{z}_\texttt{n}/\texttt{x}_\texttt{n}] \\ \Gamma, \texttt{z}_1 : \texttt{T}_1 ... \texttt{z}_\texttt{n} : \texttt{T}_\texttt{n} \vdash \texttt{stmts}' \ : \ \texttt{T}' \\ \Gamma \vdash \texttt{T}' \leq_{wdn} \texttt{T}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{mBody} \ : \ \texttt{T}_1 \times ... \times \texttt{T}_\texttt{n} \rightarrow \texttt{T} \\ \mathcal{C}\{\!\{\Gamma, \texttt{mBody}\}\!\} = \ \texttt{m is } \lambda \texttt{x}_1 : \texttt{T}_1 ... \lambda \texttt{x}_\texttt{n} : \texttt{T}_\texttt{n}.\{\mathcal{C}\{\!\{\Gamma, \texttt{stmts}\}\!\}\}\end{array}}$$

$$\frac{\begin{array}{l}\texttt{n}, \texttt{k}, \texttt{l} \geq 0, \quad \Gamma \vdash \diamond \\ \Gamma(\texttt{C}) = \texttt{C ext C}' \ \texttt{impl I}_1 ... \texttt{I}_\texttt{n}\{\texttt{v}_1 : \texttt{T}_1 ... \texttt{v}_\texttt{k} : \texttt{T}_\texttt{k}, \texttt{m}_1 : \texttt{MT}_1 ... \texttt{m}_\texttt{l} : \texttt{MT}_\texttt{l}\} \\ \texttt{cBody} = \texttt{C ext C}' \ \{\texttt{mBody}_1, ... \texttt{mBody}_\texttt{l}\} \\ \Gamma(\texttt{this} : \texttt{C}) = \texttt{Undef} \\ \texttt{mBody}_\texttt{i} = \texttt{m}_\texttt{i} \texttt{ is mPrsSts}_\texttt{i} \quad \texttt{i} \in \{1...\texttt{l}\} \\ \Gamma, \texttt{this} : \texttt{C} \vdash \texttt{mBody}_\texttt{i} \ : \ \texttt{MT}_\texttt{i} \quad \texttt{i} \in \{1...\texttt{l}\}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{cBody} \diamond \\ \mathcal{C}\{\!\{\Gamma, \texttt{cBody}\}\!\} = \ \texttt{C ext C}' \ \{\mathcal{C}\{\!\{\Gamma, \texttt{this} : \texttt{C}, \texttt{mBody}_1\}\!\} ... \mathcal{C}\{\!\{\Gamma, \texttt{this} : \texttt{C}, \texttt{mBody}_\texttt{l}\}\!\}\}\end{array}}$$

$$\frac{\begin{array}{l}\texttt{n} \geq 0, \quad \texttt{p} = \texttt{cBody}_1, ... \texttt{cBody}_\texttt{n} \\ \texttt{cBody}_\texttt{i} = \texttt{C ext } ..., \texttt{cBody}_\texttt{j} = \texttt{C ext } ... \\ \qquad \Longrightarrow \quad \texttt{i} = \texttt{j} \quad \texttt{i}, \texttt{j} \in \{1...\texttt{n}\} \\ \Gamma \vdash \texttt{cBody}_\texttt{i} \diamond \quad \texttt{i} \in \{1...\texttt{n}\}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{p} \diamond \\ \mathcal{C}\{\!\{\Gamma, \texttt{p}\}\!\} = \mathcal{C}\{\!\{\Gamma, \texttt{cBody}_1\}\!\} ... \mathcal{C}\{\!\{\Gamma, \texttt{cBody}_\texttt{n}\}\!\}\end{array}} \qquad \frac{\begin{array}{l}\mathit{Classes}(\Gamma) = \mathit{Classes}(\texttt{p}) \\ \Gamma \vdash \texttt{p} \diamond\end{array}}{\Gamma \vdash \texttt{p} \diamond\!\!\!\diamond}$$

Fig. 10. types for Java$_s$ method bodies, class bodies, and program bodies

A program $\texttt{p} = \texttt{cBody}_1, ... \texttt{cBody}_\texttt{n}$ is well-formed, *i.e.* $\Gamma \vdash \texttt{p} \diamond$, if it contains no more than one class body for each identifier, and if all class bodies, $\texttt{cBody}_\texttt{i}$, are well-typed and satisfy their declarations. Furthermore, each class is transformed by $\mathcal{C}$. Finally, as described in the last rule in figure 10, a program is complete, *i.e.* $\Gamma \vdash \texttt{p} \diamond\!\!\!\diamond$, if it is well-formed and provides a class body for each of the classes declared in the environment Γ.

The following two functions will be needed for the operational semantics. In a class body cBody the function *MethBody*(m, AT, cBody) finds the method body with identifier m and argument types AT, if it exists. From the requirements for classes in figure 10, it follows that for a well-formed environment Γ, the function *MethBody*(m, AT, cBody) returns either an empty set or a set with one element. In a program p the function *MethBody*(m, AT, C, p) finds the method body with identifier m and argument types AT, in the nearest superclass of class C – if it exists. It returns a single pair consisting of the class with the appropriate method body, and the method body itself or the empty set if none exists.

Definition 8 *Given a class body* $\mathtt{cBody} = \mathtt{C\ ext\ C'}\ \{\mathtt{mBody_1}, ...\mathtt{mBody_n}\}$, *argument types* $\mathtt{AT}$, *and a program* $\mathtt{p}$, *we define* method look up *as follows:*

- $\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{cBody}) = \{\ \mathtt{mBody_j} \mid \mathtt{mBody_j} = \mathtt{m\ is}\ \ \lambda \mathtt{x_1} : \mathtt{T_1} ... \ \lambda \mathtt{x_k} : \mathtt{T_k}.\{...\}$
  and $\mathtt{AT} = \mathtt{T_1} \times ... \times \mathtt{T_k}\ \}$
- $\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{Object}, \mathtt{p}) = \emptyset$
- $\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{C}, \mathtt{p}) = (\mathtt{C}, \mathtt{mBody})$
  if $\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{cBody}) = \{\mathtt{mBody}\}$, *where* $\mathtt{cBody} = \mathtt{p}(\mathtt{C})$
- $\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{C}, \mathtt{p}) = \mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{C'}, \mathtt{p})$
  if $\mathit{MethBody}(\mathtt{m}, \mathtt{AT}, \mathtt{cBody}) = \emptyset$, *where* $\mathtt{p}(\mathtt{C}) = \mathtt{C\ ext\ C'}...$

In figure 11 we define the type rules for exceptions. A `throw` statement has the type `void` if the expression following the `throw` indicates an exception. Similarly, the `try` ... `catch` ... `finally` statements have the type `void`, provided that the constituent statement lists are well-typed, and that the names of exception classes and new variables appear after each `catch`. The additional Java requirements, that no class $\mathtt{E_i}$ should appear more than once, and that no class should appear preceded by a subclass are expressed in [21] but are omitted here, since they do not affect the subject reduction property.

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{E}, \quad \mathtt{e} \neq \iota_\mathtt{i} \qquad \Gamma \vdash \mathtt{E} \sqsubseteq \mathtt{Exception}}{\Gamma \vdash \mathtt{throw\ e} : \mathtt{void}}$$

$$\mathcal{C}\{\!\{\Gamma, \mathtt{throw\ e}\}\!\} = \mathtt{throw}\ \mathcal{C}\{\!\{\Gamma, \mathtt{e}\}\!\}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0, \quad \mathtt{v_i}, \mathtt{z_i} \ \text{ new in } \Gamma \quad \mathtt{i} \in \{1...\mathtt{n}\} \\ \Gamma \vdash \mathtt{E_i} \ \sqsubseteq \mathtt{Exception} \quad \mathtt{i} \in \{1...\mathtt{n}\} \\ \Gamma, \mathtt{z_i} : \mathtt{E_i} \ \vdash \mathtt{stmts_i}[\mathtt{z_i}/\mathtt{v_i}] \ : \ \mathtt{void} \quad \mathtt{i} \in \{1...\mathtt{n}\} \\ \Gamma \vdash \mathtt{stmts'} \ : \ \mathtt{void}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{try\ stmts_0\ catch\ E_1\ v_1\ stmts_1\ ...\ catch\ E_n\ v_n\ stmts_n} \ : \ \mathtt{void} \\ \Gamma \vdash \mathtt{try\ stmts_0\ catch\ E_1\ v_1\ stmts_1\ ...\ catch\ E_n\ v_n\ stmts_n} \\ \quad \mathtt{finally\ stmts'} \qquad\qquad : \ \mathtt{void}\end{array}}$$

$$\begin{array}{l}\mathcal{C}\{\!\{\Gamma, \mathtt{try\ stmts_0\ catch\ E_1\ v_1\ stmts_1\ ...\ catch\ E_n\ v_n\ stmts_n}\}\!\} \\ \quad = \ \mathtt{try}\ \mathcal{C}\{\!\{\Gamma, \mathtt{stmts_0}\}\!\}\ \mathtt{catch\ E_1\ v_1}\ \mathcal{C}\{\!\{\Gamma, \mathtt{stmts_1}\}\!\}\ ... \\ \qquad \mathtt{catch\ E_n\ v_n}\ \mathcal{C}\{\!\{\Gamma, \mathtt{stmts_n}\}\!\}\end{array}$$

$$\begin{array}{l}\mathcal{C}\{\!\{\Gamma, \mathtt{try\ stmts_0\ catch\ E_1\ v_1\ stmts_1\ ...\ catch\ E_n\ v_n\ stmts_n\ finally\ stmts'}\}\!\} \\ \quad = \ \mathtt{try}\ \mathcal{C}\{\!\{\Gamma, \mathtt{stmts_0}\}\!\}\ \mathtt{catch\ E_1\ v_1}\ \mathcal{C}\{\!\{\Gamma, \mathtt{stmts_1}\}\!\}\ ... \\ \qquad \mathtt{catch\ E_n\ v_n}\ \mathcal{C}\{\!\{\Gamma, \mathtt{stmts_n}\}\!\}\ \mathtt{finally}\ \mathcal{C}\{\!\{\Gamma, \mathtt{stmts'}\}\!\}\end{array}$$

Fig. 11. Java$_s$ types for exceptions

5.1 Properties of the Java$_s$ type system

The following lemma says that the Java$_s$ type system is deterministic, and that in a complete 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 3 *For types* $\mathtt{T}$, $\mathtt{T}'$ *and Java*$_s$ *term* $\mathtt{t}$:

- *If* $\Gamma \vdash \mathtt{t} : \mathtt{T}$ *and* $\Gamma \vdash \mathtt{t} : \mathtt{T}'$ *then* $\mathtt{T} = \mathtt{T}'$.
- *If* $\Gamma \vdash \mathtt{t} : \mathtt{T}$ *then* $\Gamma, \sigma \vdash \mathcal{C}\{\Gamma, \mathtt{t}\} : \mathtt{T}$

For any well-formed environment Γ, *variable types* $\mathtt{T}$, $\mathtt{T}_1$, ...,$\mathtt{T}_n$, $\mathtt{T}_{n+1}$, *class* $\mathtt{C}$, *Java*$_s$ *program* $\mathtt{p}$, *with* $\Gamma \vdash \mathtt{p} \ll$:

- *If* $\mathtt{p} \vdash \mathtt{C} \sqsubseteq \mathtt{C}'$ *then* $\Gamma \vdash \mathtt{C} \leq_{wdn} \mathtt{C}'$

Furthermore, if

- $\Gamma \vdash \mathtt{C} \leq_{wdn} \mathtt{T}$
- $\mathtt{T}_1 \times ... \mathtt{T}_n \to \mathtt{T}_{n+1} \in \mathit{MSigs}(\Gamma, \mathtt{T}, \mathtt{m})$

then $\exists \mathtt{T}'_{n+1}, \mathtt{C}'$:

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

5.2 Absence of the 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 $\leq_{wdn}$ relation, it would have had the form:

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

The type system introduced in this paper does not obey the subsumption rule. For instance, the type of `aPhil.like` is `Phil`, but the type of `pascal.like` is `Food`, though $\Gamma_0 \vdash$ `aPhil` : `Phil`, $\Gamma_0 \vdash$ `pascal` : `FrPhil`, and $\Gamma_0 \vdash$ `FrPhil` $\leq_{wdn}$ `Phil`. In fact, the introduction of the subsumption rule would make this type system non-deterministic – although [20] 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.

$$\frac{\Gamma \vdash_{se} \mathtt{v} \ : \ \mathtt{T} \quad \Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{C} \quad \mathit{FDec}(\Gamma, \mathtt{C}, \mathtt{f}) = (\ \mathtt{C}, \mathtt{T}')}{\Gamma \vdash_{se} \mathtt{v}.[\mathtt{C}]\mathtt{f} \ : \ \mathtt{T}'}$$

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

$$\frac{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C} \quad \forall \mathtt{f}, \mathtt{C}', \mathtt{T}' \text{ with } (\mathtt{C}', \mathtt{T}') \in \mathit{FDecs}(\Gamma, \mathtt{C}, \mathtt{f}) : \ \exists \mathtt{i} \in \{\mathtt{1...n}\} : \mathtt{f_i} = \mathtt{f}, \mathtt{C_i} = \mathtt{C}', \mathtt{T_i} = \mathtt{T}' \quad \Gamma \vdash_{se} \mathtt{v_i} \ : \ \mathtt{T_i} \mathtt{i} \in \{\mathtt{1...n}\}}{\Gamma \vdash_{se} \mathtt{new\ C} \ll \mathtt{f_1\ C_1\ v_1, ...f_n\ C_n\ v_n} \gg \ : \ \mathtt{C}}$$

$$\frac{\mathtt{n} \geq 1, \mathtt{k} \geq 0 \quad \Gamma \vdash \mathtt{T} \diamond_{VarType}, \quad \mathit{NOT}\ \ \Gamma \vdash \mathtt{T} \leq \mathtt{T} \quad \Gamma \vdash_{se} \mathtt{e_i} \ : \ \mathtt{int} \quad \mathtt{i} \in \{\mathtt{1...n}\} \quad \Gamma \vdash_{se} \mathtt{v} \ : \ \mathtt{T}}{\Gamma \vdash_{se} \mathtt{new\ T}[\mathtt{e_1}]...[\mathtt{e_n}][]_1...[]_k[\![\mathtt{v}]\!] \ : \ \mathtt{T}[]_1...[]_{n+k}}$$

Fig. 12. differences between Java$_{se}$ types and Java$_s$ types

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

After giving types to Java$_s$ terms, we also give types to Java$_{se}$ terms. However, the rationale for typing the two languages is different: Java$_s$ typing corresponds to typing performed by a Java compiler, and it determines whether a term is well-formed. Java$_{se}$ typing, on the other hand, does not correspond to type checking actually performed, it is needed in order to express the subject reduction theorem. A Java$_{se}$ term that has emerged by enriching a well-typed Java$_s$ term will be well-typed too, and will have the same type as the latter, *c.f.* lemma 5. Therefore, the Java$_{se}$ type rules correspond to Java$_s$ type rules, except where the expressions have different syntax.

Figure 12 contains the four cases where Java$_{se}$ syntax differs from that of Java$_s$, and therefore, where Java$_{se}$ types differ from Java$_s$ types. The assertion $\Gamma \vdash_{se} \mathtt{t} \ : \ \mathtt{T}$ signifies that the Java$_{se}$ term $\mathtt{t}$ has type $\mathtt{T}$ in the Java$_{se}$ type system. Thus, we use the subscript se on $\vdash$ in order to distinguish between type systems.

The first rule in figure 12 describes 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.* $\mathtt{C}$) instead of the type of the variable on the left of the field access (*i.e.* $\mathtt{T}$).

In the second rule we consider Java$_{se}$ method calls: we search for appropriate methods using the descriptor signature, $(\mathtt{T_2} \times ... \times \mathtt{T_n})$, instead of the types of the actual expressions, $(\mathtt{T}'_2, ...\mathtt{T}'_n)$. 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 9 *For environment Γ, identifier* $\mathtt{m}$*, type* $\mathtt{T}_1$*, argument types* $\mathtt{AT}$*, we define:*
$\mathit{FirstFit}(\Gamma, \mathtt{m}, \mathtt{T}_1, \mathtt{AT}) = \{(\mathtt{T}, \mathtt{MT}) \mid (\mathtt{T}, \mathtt{MT}) \in \mathit{MDecs}(\Gamma, \mathtt{T}_1, \mathtt{m})\ \mathit{and}\ \mathit{Args}(\mathtt{MT}) = \mathtt{AT}\}$

The last two rules in figure 12 describe object and array creation. The additional requirement over Java$_s$ is that initialization values be of the appropriate type.

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

The following lemma states that no more than one signature with argument types AT can be found for a type T. This signature will always be found in a superclass or superinterface of T. Also, once such a signature is found, the same signature can be found for any subclass or subinterface of T.

Lemma 4 *For a well-formed environment Γ, types* T, T′, T″*, and argument types* AT*:*

- $card(FirstFit(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{AT})) \leq 1$
- $\exists \mathtt{MT} : FirstFit(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{AT}) = \{(\mathtt{T}', \mathtt{MT})\} \implies \Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'$
- $\exists \mathtt{MT} : FirstFit(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{AT}) = \{(\mathtt{T}', \mathtt{MT})\}$ *and* $\Gamma \vdash \mathtt{T}'' \leq_{wdn} \mathtt{T}$
  $\implies \exists \mathtt{T}''' : FirstFit(\Gamma, \mathtt{m}, \mathtt{T}', \mathtt{AT}) = (\mathtt{T}''', \mathtt{MT})$ *and* $\Gamma \vdash \mathtt{T}''' \leq_{wdn} \mathtt{T}'$

Not surprisingly, a well-typed Java$_s$ expression of type T is enriched into a Java$_{se}$ expression which has the type T as well.

Lemma 5 *For type* T*, environment Γ, Java$_s$ term* t*:*

$$\Gamma \vdash \mathtt{t} : \mathtt{T} \implies \Gamma \vdash_{se} \mathcal{C}\{\!\{\Gamma, \mathtt{t}\}\!\} : \mathtt{T}$$

7 Java$_r$ – the run-time language

As we have seen in section 4, Java$_{se}$ is an enriched version of Java$_s$ containing compile-time type information necessary for execution. However, at run-time, terms may be created, whose syntax is not described in Java$_{se}$. For this, we extend Java$_{se}$ obtaining Java$_r$, the run-time language, whose syntax is given in figure 13. Java$_r$ is a pure superset of Java$_{se}$, and it corresponds to Don Syme's Java$_R$ from chapter 4, with the difference that Java$_r$ also allows for exceptions. Java$_r$ is a superset of Java$_{light}$ from chapter 5, because Java$_{light}$ does not describe additional artifacts that may arise at run-time only.

The additional artifacts, that may appear at run-time and are not part of Java$_{se}$ but are part of Java$_r$, arise through addresses and the `null` value. Addresses have the form $\iota_{\mathtt{i}}$; they represent references to objects and arrays, and may appear wherever a value is expected, as well as in array and field accesses. Therefore, Java$_r$ instance variables may have the form $\iota_{\mathtt{i}}$.[`ClassName`]`VarName`, or $\iota_{\mathtt{i}}$[*Expr*], and expressions may have the form $\iota_{\mathtt{i}}$. An access to `null` may arise during evaluation of array or field access variables, therefore Java$_r$ expressions may have the form `null`.[`ClassName`]`VarName`, or `null`[*Expr*].

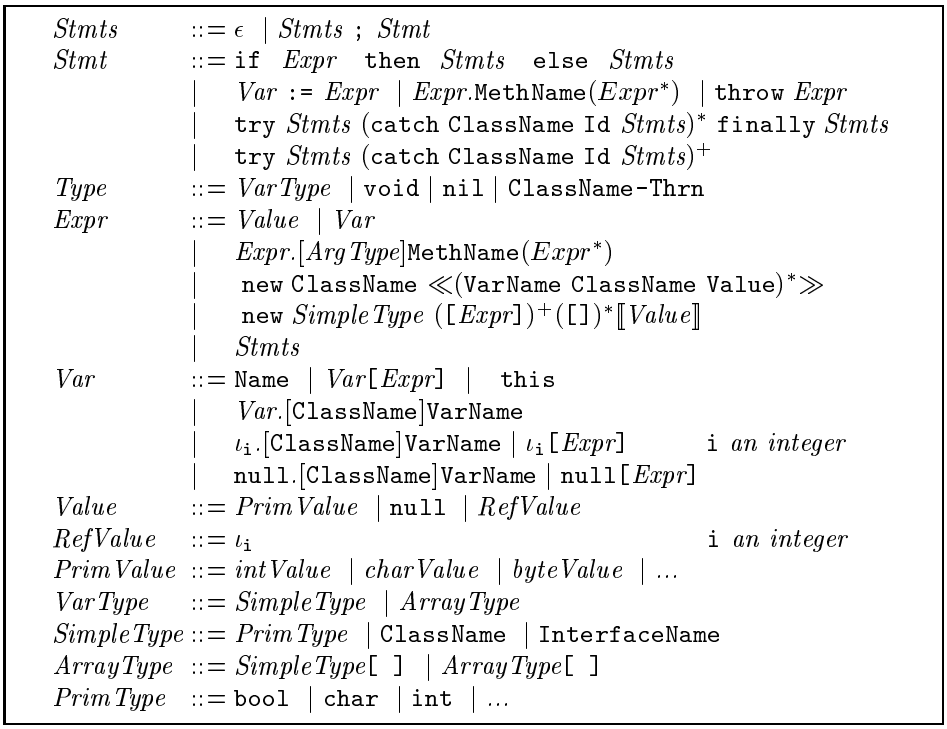


Fig. 13. Java$_r$ expressions

8 The operational semantics

Figure 14 describes the run-time model for the operational semantics. For a given program p, the operational semantics maps configurations to new configurations. Configurations are tuples of Java$_r$ terms and states, or just states. The operational semantics is a mapping from programs and configurations to configurations.

The state is flat; it consists of mappings from identifiers to primitive values or to references, and from references to objects or arrays. Note that references may point to objects, or arrays, but they may not point to other references, primitive values, or null– this is so, because pointers in Java are implicit, and there are no pointers to pointers.

An object is annotated by its class, and it 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, *c.f.* [16] ch. 9.5. For the philosophers example, $\ll$ like Phil: ι_2, like FrPhil: null $\gg^{\texttt{FrPhil}}$ is an object of class FrPhil. It inherits the field like from Phil, and has the field like from FrPhil.

The following state σ_0 contains mappings according to the philosophers ex-

ample:

$\sigma_0(\texttt{aPhil}) = \iota_1$
$\sigma_0(\texttt{oyster}) = \iota_3$
$\sigma_0(\iota_1) = \ll\texttt{like Phil: } \iota_2\texttt{, like FrPhil: null}\gg^{\texttt{FrPhil}}$
$\sigma_0(\iota_2) = \ll\ldots\gg^{\texttt{Truth}}$
$\sigma_0(\iota_3) = \ll\ldots\gg^{\texttt{Food}}$

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.

Configuration	::=	$\langle \text{Java}_r \text{ term}, \textit{state}\rangle \cup \langle \textit{state}\rangle$
$\rightsquigarrow$	:	Java_r program $\longrightarrow$ *Configuration* $\longrightarrow$ *Configuration*
$\rightsquigarrow_{\texttt{p}}$	:	*Configuration* $\longrightarrow$ *Configuration*
State	::=	(`Ident` $\longrightarrow$ *Value*)* $\cup$ (*RefValue* $\longrightarrow$ *ObjectOrArray*)*
ObjectOrArray	::=	*Object* $\mid$ *Array*
Object	::=	$\ll$(`LabelName ClassName` : *Value*)$^*\gg^{\texttt{ClassName}}$
Array	::=	$[\![(\textit{Value})^*]\!]^{\textit{ArrayType}}$

Fig. 14. Java_r run-time model

8.1 State, object operations, ground terms

We now define operations on objects, arrays and states. These operations are well-defined, only if the object, array or state "conforms" to the types expected by the environment, a requirement that will be introduced in definition 13.

Definition 10 *For object* $\texttt{obj} = \ll \texttt{l}_1\ \texttt{C}_1 : \texttt{val}_1, \texttt{l}_2\ \texttt{C}_2 : \texttt{val}_2, \ldots, \texttt{l}_\texttt{n}\ \texttt{C}_\texttt{n} : \texttt{val}_\texttt{n} \gg^{\texttt{C}'}$, *state* σ, *value* `val`, *reference* $\iota_\texttt{i}$, *identifier or reference* `z`, *class* `C`, *field identifier* `f`, *integers* `m`, `k` *with* $\texttt{m} \geq 0$, *array* $\texttt{arr} = [\![\texttt{val}_0, \ldots \texttt{val}_{\texttt{n}-1}]\!]^{\texttt{T}[]_1 \ldots []_\texttt{m}}$, *we define:*

- *the* access to field `f` declared in class `C` *as* `obj(f,C)`:

  $\texttt{obj}(\texttt{f}, \texttt{C}) = \texttt{val}_\texttt{i}$ *if* $\texttt{f} = \texttt{l}_\texttt{i}$ *and* $\texttt{C} = \texttt{C}_\texttt{i}$

- *the* access to component `f`, `C` of an object *stored at reference* $\iota_\texttt{i}$, *in state* σ :

  $\sigma(\iota_\texttt{i}, \texttt{f}, \texttt{C}) = \sigma(\iota_\texttt{i})(\texttt{f}, \texttt{C})$

- *the* access to the k^{th} component *of* `arr`, $\texttt{arr}[\texttt{k}]$:

  $\texttt{arr}[\texttt{k}] = \texttt{val}_\texttt{k}$ *if* $0 \leq \texttt{k} \leq \texttt{n} - 1$

– *a new state,* $\sigma' = \sigma[\mathtt{z} \mapsto \mathtt{val}]$*, such that:*

$$\begin{array}{ll} \sigma'(\mathtt{z}) & = \mathtt{val} \\ \sigma'(\mathtt{z}') & = \sigma(\mathtt{z}') \quad \textit{for } \mathtt{z}' \neq \mathtt{z} : \end{array}$$

– *a new object,* $\mathtt{obj}' = \mathtt{obj}[\mathtt{f}, \mathtt{C} \mapsto \mathtt{val}]$*, a new state,* $\sigma' = \sigma[\iota_{\mathtt{i}}, \mathtt{f}, \mathtt{C} \mapsto \mathtt{val}]$:

$$\begin{array}{lll} \mathtt{obj}'(\mathtt{f}, \mathtt{C}) & = \mathtt{val} & \\ \mathtt{obj}'(\mathtt{f}', \mathtt{C}') & = \mathtt{obj}(\mathtt{f}', \mathtt{C}') & \textit{if } \mathtt{f} \neq \mathtt{f}' \textit{ or } \mathtt{C} \neq \mathtt{C}' \\ \sigma' & = \sigma[\iota_{\mathtt{i}} \mapsto \sigma(\iota_{\mathtt{i}})[\mathtt{f}, \mathtt{C} \mapsto \mathtt{val}]] & \end{array}$$

– *a new array,* $\mathtt{arr}' = \mathtt{arr}[\mathtt{k} \mapsto \mathtt{val}]$*, and a new state,* $\sigma' = \sigma[\iota_{\mathtt{i}}, \mathtt{k} \mapsto \mathtt{val}]$:

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

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

Definition 11 *A Java$_r$ term* `t` *is*

– ground *iff* `t` *is a primitive value, or* `t=null` *, or* $\mathtt{t} = \iota_{\mathtt{i}}$ *for some* `i`*;*

– l-ground *iff* `t=id` *for some identifier* `id`*, or* `t=` $\iota_{\mathtt{i}}$`.[C]f` *for a class* `C` *and a field* `f` `t= null.[C]f` *, or* $\mathtt{t} = \iota_{\mathtt{i}}[\mathtt{k}]$ *or* $\mathtt{t} = \mathtt{null}\ [\mathtt{k}]$ *for some integer* `k`*.*

8.2 Program execution

Figures 15, 16, 17, 18, and 19 describe rewriting of Java$_r$ terms. We chose small step semantics because we found this more intuitive. Interestingly, it turns out that large step semantics allow for a simpler proof of subject reduction, and in particular, do not require different type rules for Java$_r$ assignment to array components and the other assignments statements [23]. On the other hand, small steps allow the description of co-routines [21]. In figure 15 we describe the evaluation of variables, field and array access, and the creation of new objects or arrays.

Variables (*i.e.* identifiers, instance variable access or array access) are evaluated from left to right. The rules about assignment in figure 17 prevent an expression like `x`, or $\iota_{\mathtt{i}}$`[C]v`, appearing on the left hand side of an assignment from being rewritten further. They allow an expression of the form `u[C1].w[C2].x[C3].y` to be rewritten to an expression of the form $\iota_{\mathtt{j}}$`[C3].y` for some `j`. Furthermore, there is *no* rule of the form $\langle \iota_{\mathtt{j}}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \sigma(\iota_{\mathtt{j}}), \sigma \rangle$. 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 [16], which require full evaluation of the expression to the left of the brackets. Thus, with our operational semantics, the term `a`$[(\mathtt{a} := \mathtt{b})[3]]$ corresponds to the term `a`$[\mathtt{b}[3]]; \mathtt{a} := \mathtt{b}$.

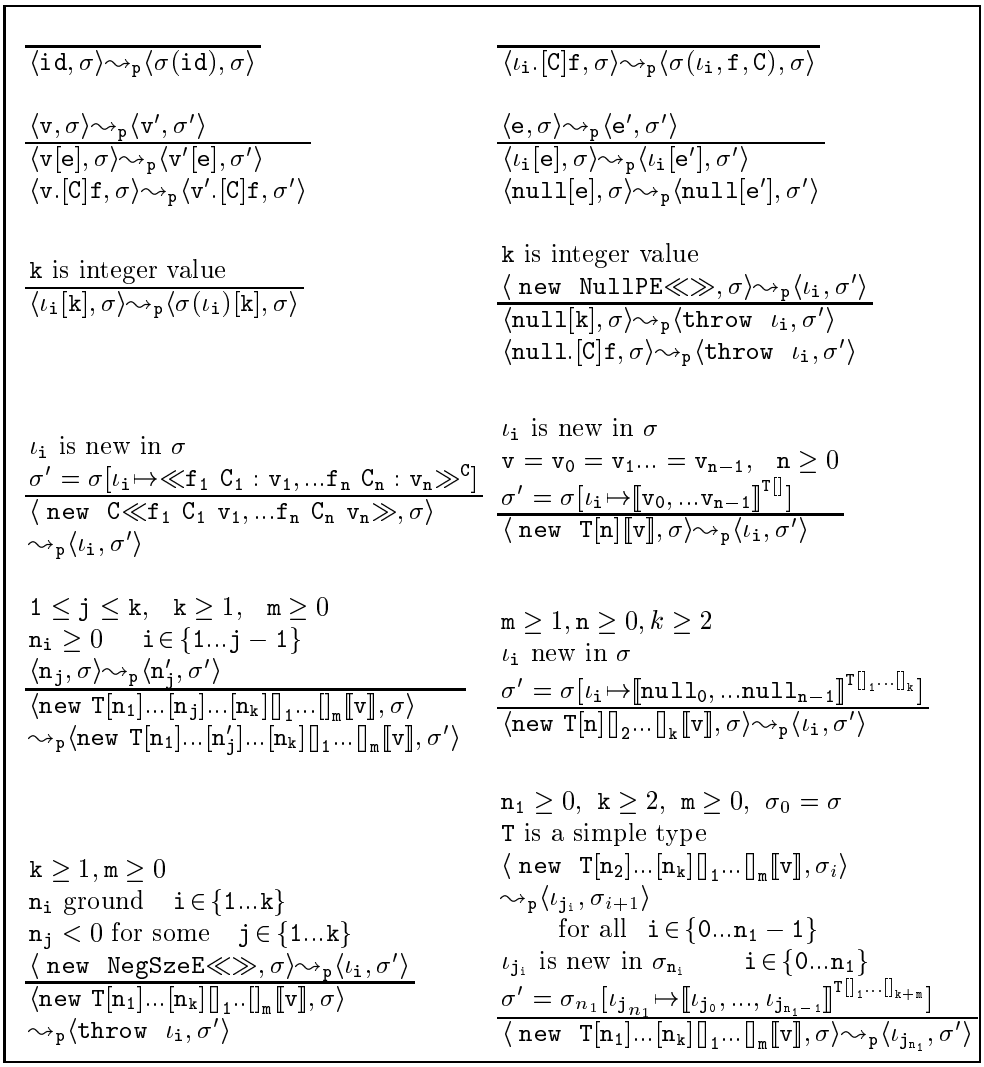


Fig. 15. expression execution

The last six rules in figure 15 describe the creation of new objects or arrays, *c.f.* ch. 15.8-15.9 of [16]. 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 of the object are assigned initial values (calculated at compile-time, cf definition 6) of the type to which *they* belong.

For example, for a state σ_{00} the expression `new int[2][3][][][0]` would be executed as: $\langle$ `new int[2][3][][][0]`$, \sigma_{00}\rangle \hookrightarrow_{p'} \langle \iota_7, \sigma_{01}\rangle$ where ι_5, ι_6, and ι_7 are new

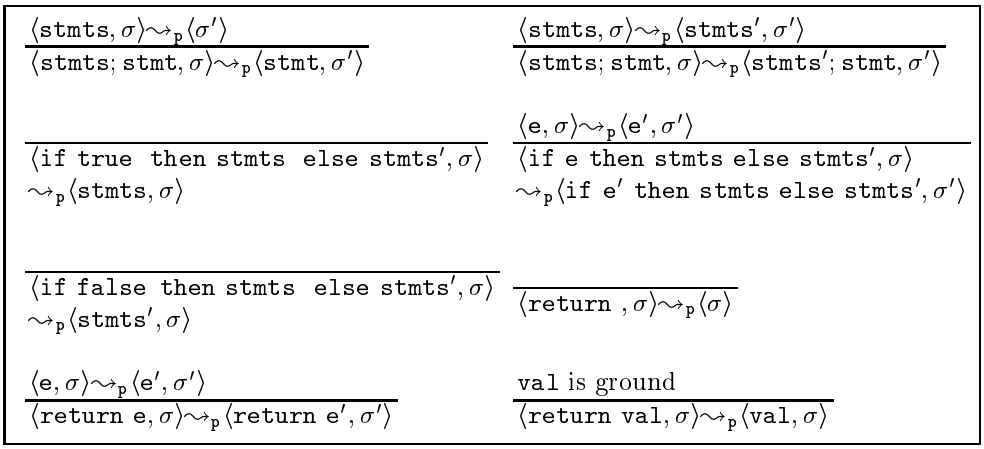


Fig. 16. statement execution

in σ_{00}, and they have the following contents in σ_{01}:
$\sigma_{01}(\iota_5) = [\![\texttt{null}, \texttt{null}, \texttt{null}]\!]^{\texttt{int}[][][]}$
$\sigma_{01}(\iota_6) = [\![\texttt{null}, \texttt{null}, \texttt{null}]\!]^{\texttt{int}[][][]}$
$\sigma_{01}(\iota_7) = [\![\iota_5, \iota_6]\!]^{\texttt{int}[][][][]}$

Figure 16 describes statement execution. 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 branch is executed. A `return` statement terminates execution. A statement returning an expression evaluates this expression until ground and replaces itself by this ground value – thus modeling methods returning values.

Figure 17 describes the evaluation of assignments. According to the first rule, the left hand side is evaluated first, until it becomes l-ground. Then, according to the next rule, the right hand side of the assignment is evaluated, up to the point of obtaining a ground term. Assignment to variables or to object components modifies the state accordingly.

The last three rules describe assignment to array components where the index being within bounds has to be checked first (if not, `IndOutBndE` is thrown), then the value has to fit the array (if not, `ArrStoreE` is thrown), and, if the two above requirements are satisfied, then the assignment is performed. Fitting, a requirement which ensures that an object or array value is of a type that can be appropriately stored into another array, is described in definition 12.

Other exceptions (*e.g.* null access) need not be considered in these rules, because they would be checked by the variable rules from figure 16, and then propagated by the exception rules from figure 19. Also, we have *no* rule of the form $\langle \iota_j := \texttt{value}, \sigma\rangle \rightsquigarrow_p \ldots$ because overwriting of objects is not possible in Java

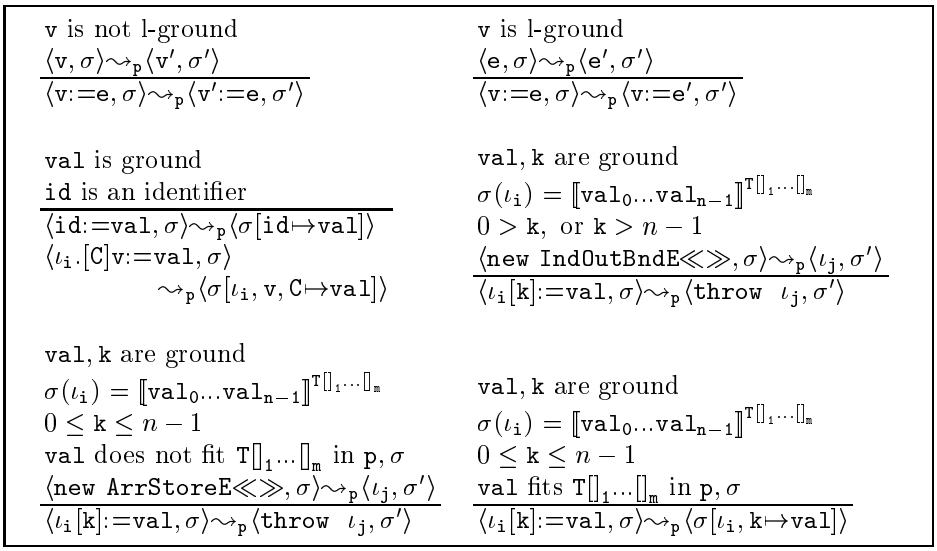


Fig. 17. assignment execution

– only sending messages to them, or overwriting selected instance variables.

Definition 12 *A value* $\mathtt{val}$ fits *a type* $\mathtt{T} = \mathtt{T}'[]$ *in a program* $\mathtt{p}$*, iff* $\mathtt{val}$ *is primitive, or* $\mathtt{val} = \mathtt{null}$ *, or* $\sigma(\mathtt{val}) = \ll ... \gg^{\mathtt{C}}$ *and* $\mathtt{p} \vdash \mathtt{C} \sqsubseteq \mathtt{T}'$*, or* $\sigma(\mathtt{val}) = [\![...]\!]^{\mathtt{T}''}$ *and* $\mathtt{p} \vdash \mathtt{T}'' \sqsubseteq \mathtt{T}'$.

Primitive values fit any array type, *e.g.* 4 fits the type `FrPhil[][][]`. We defined the "fits" relation this way, because run-time checks for assignments to array components are only necessary when the value on the right hand side is of class type, and *not* when it is a primitive value, *c.f.* lemma 11. Also, note that in the above definition the types $\mathtt{T}'$ and $\mathtt{T}''$ may be array types themselves, and that the subclass relationship is monotonic with the array type constructor (*i.e.* $\mathtt{p} \vdash \mathtt{C} \sqsubseteq \mathtt{C}'$ implies that $\mathtt{p} \vdash \mathtt{C}[] \sqsubseteq \mathtt{C}'[]$).

Figure 18 describes the evaluation of method calls. The receiver and argument expressions are evaluated left to right, *c.f.* ch. 9.3 in [19]. The first rule describes rewriting the $\mathtt{k}^{th}$ expression, where all the previous expressions (*i.e.* $\mathtt{val_i}, \mathtt{i} \in \{1...\mathtt{k}-1\}$) are ground. The second rule requires the exception `NullPE` to be thrown if the receiver is `null`. The third rule describes dynamic method look up, taking into account the argument types, and the statically calculated method descriptor `AT`. The term $\mathtt{t}[\mathtt{t}'/\mathtt{x}]$ has the usual meaning of replacing the variable $\mathtt{x}$ by the term $\mathtt{t}'$ in the term $\mathtt{t}$.

Execution of the method call `aPhil.[Phil]think (aPhil)` results in the following rewrites:

$$\frac{\texttt{val}_\texttt{i} \text{ is ground for } \texttt{i} \in \{1...\texttt{k}-1\}, \texttt{n} \geq \texttt{k} \geq 1 \quad \langle \texttt{e}_\texttt{k}, \sigma\rangle \rightsquigarrow_\texttt{p} \langle \texttt{e}'_\texttt{k}, \sigma'\rangle}{\langle \texttt{val}_1.[\texttt{AT}]\texttt{m}(\texttt{val}_2, ..., \texttt{val}_{\texttt{k}-1}, \texttt{e}_\texttt{k}, ...\texttt{e}_\texttt{n}), \sigma\rangle \rightsquigarrow_\texttt{p} \langle \texttt{val}_1.[\texttt{AT}]\texttt{m}(\texttt{val}_2, ..., \texttt{val}_{\texttt{k}-1}, \texttt{e}'_\texttt{k}, ...\texttt{e}_\texttt{n}), \sigma'\rangle}$$

$$\frac{\texttt{val}_\texttt{i} \text{ is ground for } \texttt{i} \in \{2...\texttt{n}\}, \texttt{n} \geq 1}{\langle \texttt{null}.[\texttt{AT}]\texttt{m}(\texttt{val}_2, ...\texttt{val}_\texttt{n}), \sigma\rangle \rightsquigarrow_\texttt{p} \langle \texttt{throw new NullPE}\ll\gg, \sigma\rangle}$$

$$\frac{\begin{array}{l}\texttt{n} \geq 1 \\ \texttt{val}_\texttt{i} \text{ is ground for } \texttt{i} \in \{1...\texttt{n}\} \\ \sigma(\texttt{val}_1) = \ll ... \gg^\texttt{C} \\ \texttt{AT} = \texttt{T}_2 \times ... \times \texttt{T}_\texttt{n} \\ \mathit{MethBody}(\texttt{m}, \texttt{AT}, \texttt{C}, \texttt{p}) = (\texttt{C}', \texttt{m is } \lambda \texttt{x}_2 : \texttt{T}_2 ... \lambda \texttt{x}_\texttt{n} : \texttt{T}_\texttt{n}.\{\texttt{stmts}\}) \\ \texttt{z}_\texttt{i} \text{ are new identifiers in } \sigma \\ \sigma' = \sigma[\texttt{z}_1 \mapsto \texttt{val}_1] ... [\texttt{z}_\texttt{n} \mapsto \texttt{val}_\texttt{n}] \\ \texttt{stmts}' = \texttt{stmts}[\texttt{z}_1/\texttt{this}, \texttt{z}_2/\texttt{x}_2, ...\texttt{z}_\texttt{n}/\texttt{x}_\texttt{n}]\end{array}}{\langle \texttt{val}_1.[\texttt{AT}]\texttt{m}(\texttt{val}_2, ...\texttt{val}_\texttt{n}), \sigma\rangle \rightsquigarrow_\texttt{p} \langle \texttt{stmts}', \sigma'\rangle}$$

Fig. 18. evaluation of method call

$\langle \texttt{aPhil}.[\texttt{Phil}]\texttt{think}(\texttt{aPhil}), \sigma_0\rangle \rightsquigarrow_{\texttt{p}'} \langle \iota_1.[\texttt{Phil}]\texttt{think}(\texttt{aPhil}), \sigma_0\rangle \rightsquigarrow_{\texttt{p}'}$
$\langle \iota_1.[\texttt{Phil}]\texttt{think}(\iota_1), \sigma_0\rangle \rightsquigarrow_{\texttt{p}'} \langle (\texttt{w}.[\texttt{FrPhil}]\texttt{like:=oyster}; ...), \sigma_1\rangle \rightsquigarrow_{\texttt{p}'}$
$\langle (...), \sigma_2\rangle$

where σ_1, σ_2 are:

$\sigma_1(\texttt{aPhil}) = \sigma_0(\texttt{aPhil}) = \iota_1$
$\sigma_1(\texttt{oyster}) = \sigma_0(\texttt{oyster}) = \iota_3$
$\sigma_1(\texttt{w}) = \iota_1$
$\sigma_1(\texttt{w}') = \iota_1$
$\sigma_1(\iota_1) = \sigma_0(\iota_1) = \ll \texttt{like Phil: } \iota_2\texttt{, like FrPhil: null} \gg^{\texttt{FrPhil}}$
$\sigma_1(\iota_2) = \sigma_0(\iota_2) = \ll ... \gg^{\texttt{Truth}}$
$\sigma_1(\iota_3) = \sigma_0(\iota_3) = \ll ... \gg^{\texttt{Food}}$
$\sigma_2(\texttt{z}) = \sigma_1(\texttt{z}) \quad \forall \texttt{z} \neq \iota_1$
$\sigma_2(\iota_1) = \ll \texttt{like Phil: } \iota_2\texttt{, like FrPhil: } \iota_3 \gg^{\texttt{FrPhil}}$

The rules in figure 19 describe the operational semantics for propagation and handling exceptions. Thus, $\langle \texttt{try throw new E}; \texttt{f}(\texttt{x})...\texttt{catch E}_1\ \texttt{v}_1\ \texttt{stmts}_1, \sigma\rangle$ would rewrite to $\langle \texttt{try throw } \iota_\texttt{i}\ \texttt{catch E}_1\ \texttt{v}_1\ \texttt{stmts}_1, \sigma\rangle$ then to $\langle \texttt{stmts}'_1, \sigma\rangle$, if E is a subclass of $\texttt{E}_1$. During execution the term maintains its type, which is void; the subterm $\texttt{throw } \iota_\texttt{i}$ has the type E-Thrn.

$$\frac{\langle \mathtt{e},\sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{e}',\sigma'\rangle \quad \mathtt{cont}\sqsubset\cdot\sqsupset \text{ a context}}{\langle \mathtt{cont}\sqsubset\mathtt{throw\ e}\sqsupset,\sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{cont}\sqsubset\mathtt{throw\ e}'\sqsupset,\sigma'\rangle \quad \langle \mathtt{cont}\sqsubset\mathtt{throw}\ \iota_{\mathtt{i}}\sqsupset,\sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{throw}\ \iota_{\mathtt{i}},\sigma\rangle}$$

$$\frac{}{\langle \mathtt{throw\ null},\sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{throw\ new\ NullPE}\ll\gg,\sigma\rangle}$$

$$\frac{\langle \mathtt{stmts},\sigma\rangle\rightsquigarrow_{\mathtt{p}}\langle\sigma'\rangle}{\langle \mathtt{try\ stmts\ catch\ E_1\ v_1\ stmts_1\ \ldots\ catch\ E_n\ v_n\ stmts_n},\sigma\rangle\rightsquigarrow_{\mathtt{p}}\langle\sigma'\rangle \quad \langle \mathtt{try\ stmts\ catch\ E_1\ v_1\ stmts_1\ \ldots\ catch\ E_n\ v_n\ stmts_n\ finally\ stmts_{n+1}},\sigma\rangle \rightsquigarrow_{\mathtt{p}}\langle \mathtt{stmts_{n+1}},\sigma'\rangle}$$

$$\frac{\langle \mathtt{stmts},\sigma\rangle\rightsquigarrow_{\mathtt{p}}\langle \mathtt{stmts}',\sigma'\rangle}{\langle \mathtt{try\ stmts\ catch\ E_1\ v_1\ stmts_1\ \ldots\ catch\ E_n\ v_n\ stmts_n},\sigma\rangle \rightsquigarrow_{\mathtt{p}}\langle \mathtt{try\ stmts}'\ \mathtt{catch\ E_1\ v_1\ stmts_1\ \ldots\ catch\ E_n\ v_n\ stmts_n},\sigma'\rangle \quad \langle \mathtt{try\ stmts\ catch\ E_1\ v_1\ stmts_1\ \ldots\ catch\ E_n\ v_n\ stmts_n\ finally\ stmts_{n+1}},\sigma\rangle \rightsquigarrow_{\mathtt{p}}\langle \mathtt{try\ stmts}'\ \mathtt{catch\ E_1\ v_1\ stmts_1\ \ldots\ catch\ E_n\ v_n\ stmts_n\ finally\ stmts_{n+1}},\sigma'\rangle}$$

$$\frac{\sigma(\iota_{\mathtt{i}}) = \ll\ldots\gg^{\mathtt{E}} \quad \forall \mathtt{k}\in\{1\ldots\mathtt{n}\}\quad NOT\quad \mathtt{p}\vdash \mathtt{E}\sqsubseteq \mathtt{E_k}}{\langle \mathtt{try\ throw}\ \iota_{\mathtt{i}}\ \mathtt{catch\ E_1\ v_1\ stmts_1\ \ldots\ catch\ E_n\ v_n\ stmts_n},\sigma\rangle\rightsquigarrow_{\mathtt{p}}\langle \mathtt{throw}\ \iota_{\mathtt{i}},\sigma'\rangle \quad \langle \mathtt{try\ throw}\ \iota_{\mathtt{i}}\ \mathtt{catch\ E_1\ v_1\ stmts_1\ \ldots\ catch\ E_n\ v_n\ stmts_n\ finally\ stmts_{n+1}},\sigma\rangle \rightsquigarrow_{\mathtt{p}}\langle \mathtt{stmts_{n+1};\ throw}\ \iota_{\mathtt{i}},\sigma\rangle}$$

$$\frac{\begin{array}{l}\sigma(\iota_{\mathtt{i}}) = \ll\ \ldots\ \gg^{\mathtt{E}} \\ \exists \mathtt{i}\in\{1\ldots\mathtt{n}\}\ :\ \mathtt{p}\vdash \mathtt{E}\sqsubseteq \mathtt{E_i}\quad AND\qquad \forall \mathtt{k}\in\{1\ldots\mathtt{i}-1\}\quad NOT\quad \mathtt{p}\vdash \mathtt{E}\sqsubseteq \mathtt{E_k} \\ \mathtt{stmts}' = \mathtt{stmts_i}[\mathtt{z}/\mathtt{v_i}],\ \mathtt{z}\text{ new in stmts and in }\sigma \\ \sigma' = \sigma[\mathtt{z}\mapsto\iota_{\mathtt{i}}]\end{array}}{\langle \mathtt{try\ throw}\ \iota_{\mathtt{i}}\ \mathtt{catch\ E_1\ v_1\ stmts_1\ \ldots\ catch\ E_n\ v_n\ stmts_n},\sigma\rangle\rightsquigarrow_{\mathtt{p}}\langle \mathtt{stmts}',\sigma'\rangle \quad \langle \mathtt{try\ throw}\ \iota_{\mathtt{i}}\ \mathtt{catch\ E_1\ v_1\ stmts_1\ \ldots\ catch\ E_n\ v_n\ stmts_n\ finally\ stmts_{n+1}},\sigma\rangle \rightsquigarrow_{\mathtt{p}}\langle \mathtt{try\ stmts}'\ \mathtt{finally\ stmts_{n+1}},\sigma'\rangle}$$

Fig. 19. exception throwing, propagation and handling

9 Extending the type rules to Java$_r$

We gave types to Java$_{se}$ terms in order to be able to formulate a subject reduction theorem. For the same reason we shall now give types to Java$_r$ terms. The Java$_r$ type rules correspond to Java$_{se}$ type rules, except where Java$_r$ introduces new syntax, or, where necessities of the subject reduction theorem proof require otherwise.

The type of an address ($\iota_{\mathtt{i}}$) depends on the object or array pointed at in the current state σ; therefore, the type of a Java$_r$ term depends on both the environment *and* the state, and this is why Java$_r$ type assertions have the form $\Gamma,\sigma\vdash_r \mathtt{t}\ :\ \mathtt{T}$. Again, we use a subscript (in that case r) to distinguish amongst the three type systems.

Figure 20 contains the seven cases where Java$_r$ types differ from Java$_{se}$ types.

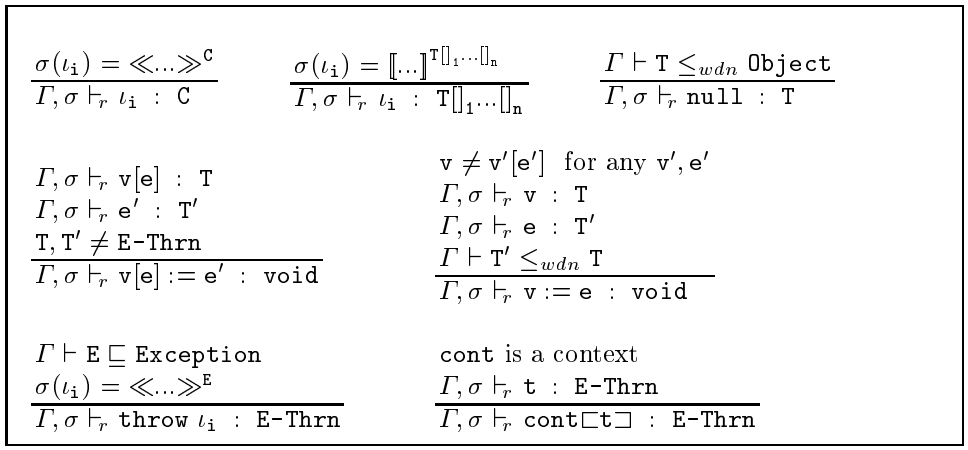


Fig. 20. differences between Java$_r$ and Java$_{se}$ types

The reasons for the differences can be classified into three categories. Firstly, those that give types to expressions that may only arise during program execution but do not involve exceptions (*i.e.* the rules for addresses and for `null`). Secondly, those that give types to terms enclosing a thrown exception (the last two rules). Thirdly, those that give types to terms that would be type-incorrect in Java$_{se}$ (*i.e.* typing of assignments, and the rules for `null`). The rules in the first category give the same type as that given if the address or `null` were replaced by an identifier of an appropriate class or array type. The rules in the second category make type-correct terms which would have been type-incorrect in Java$_{se}$. However, evaluation of such terms would not corrupt the integrity of the system, since the operational semantics requires run-time checks to be performed, and exceptions to be thrown, when certain conditions are not satisfied. The rules in the third category involve the type `C-Thrn`, a type which was not available in Java$_{se}$, or Java$_s$.

We now discuss these seven rules in more detail. The first two rules in figure 20 describe the types of addresses. If an object is stored at address $\iota_{\mathtt{i}}$, *i.e.* $\sigma(\iota_{\mathtt{i}}) = \ll\!...\!\gg^{\mathtt{C}}$, then its class, `C`, is the type of $\iota_{\mathtt{i}}$. If a `k`-dimensional array of `T` is stored at such an address, *i.e.* $\sigma(\iota_{\mathtt{i}}) = [\![...]\!]^{\mathtt{T}[]_1...[]_k}$, then $\mathtt{T}[]_1...[]_k$ is the type of this reference.

The third rule says that `null` has any reference type. This rule is required in order to be able to give a type to terms such as `null[j-4]`, which, although type-incorrect in Java$_s$, may arise during execution of Java$_r$ terms. Such terms ultimately lead to exceptions, but they do not *immediately* raise the exception `NullPE`, because the Java semantics requires other parts of the expression to be evaluated first – in our example, `j-4` has to be evaluated first. In order to be able to prove the subject reduction theorem, such expressions need a type. The

effect of this rule is, that Java$_r$ terms do not have unique types.

The fourth and fifth rules describe assignments. The Java$_r$ array assignment rule, suggested to us by Don Syme [27, 28], only requires the left hand side and the right hand side to be type-correct. It is weaker than the corresponding assignment type rule in Java$_s$or Java$_{se}$; namely, it does not require the right hand side to be of a type that can be widened to that of the left hand side. The reason for this weaker requirement is, that the type of an array may become narrower during evaluation. For example, if `z` is a one dimensional array of `Phil`, then the assignment `z[3]:=aPhil` is type-correct. However, if at run-time `z` happens to contain a reference to an array of `FrPhil`, *i.e.* $\sigma(\mathtt{z}) = \iota_{\mathtt{i}}$ and $\sigma(\iota_{\mathtt{i}}) = [\![...]\!]^{\mathtt{FrPhil}[]}$, then `z[3]:=aPhil` will be rewritten to $\iota_{\mathtt{i}}$`[3] := aPhil`. Should this term be considered type-correct? A term `y`$[3] :=$ `aPhil` would be type-incorrect if `y` were declared as an array of `FrPhil`. On the other hand, evaluation of the term $\iota_{\mathtt{i}}[3] :=$ `aPhil` will not stop here. The right hand side, in that case `aPhil`, will be evaluated, and if it returns a value which is of a subclass of `FrPhil`, then the assignment will be performed, otherwise an exception will be thrown. Therefore, in order to be able to prove subject reduction, the intermediate term $\iota_{\mathtt{i}}[3] :=$ `aPhil` has to be considered type-correct in Java$_r$. Interestingly, such a distinction between types for array assignments and other assignments is not necessary when using large steps operational semantics [23].

Finally, the last two rules in figure 20 deal with exceptions that have actually been thrown. The term `throw new E`$\ll\gg$ indicates *potential* throwing of an exception, and would be rewritten to the term `throw` $\iota_{\mathtt{i}}$, where $\iota_{\mathtt{i}}$ is the address of an object of class `E`. The latter term indicates an exception which has *actually* been thrown, and therefore has type `E-Thrn`. The *context* of an exception, defined in figure 21, encompasses all enclosing terms up to the nearest enclosing `try`... `catch` close, *i.e.* up to the first possible position at which the exception might be handled. According to the last rule in figure 20, the type of a term which is a context for a thrown exception of class `E` is `E-Thrn`. This rule allows the typing of a message expression one of whose arguments threw an exception, assignments whose left hand or right hand side threw an exception, *etc.*

9.1 Properties of the Java$_r$ type system

Trivially, any well-typed Java$_{se}$ expression retains its type for any state σ.

Lemma 6 *For type* `T`*, environment* Γ*, Java*$_{se}$ *term* `t`*, and any state* σ*:*

$$\Gamma \vdash_{se} \mathtt{t} : \mathtt{T} \implies \Gamma, \sigma \vdash_r \mathtt{t} : \mathtt{T}$$

Notice, that the opposite direction does not hold. For example, for a variable `diningFrPhils` of type `FrPhil[]`, the Java$_r$ term `diningFrPhils[3]:=aPhil` is type-correct, but the corresponding Java$_{se}$ term, `diningFrPhils[3]:=aPhil` is not. Furthermore, Java$_r$ expressions may have more than one type.

The type `E-Thrn` characterizes Java$_r$ terms that contain actually thrown exceptions. Thus, the type `E-Thrn` can only be encountered when typing Java$_r$ terms.

$$
\begin{array}{rcl}
\mathit{Context} & ::= & \mathit{ExprCont} \mid \mathit{VarCont} \mid \mathit{StmtCont} \\
\mathit{VarCont} & ::= & \mathit{VarCont}.\texttt{[ClassName]VarName} \mid \mathit{VarCont}\ \texttt{[}\mathit{Expr}\texttt{]} \\
 & \mid & \mathit{Var}\ \texttt{[}\mathit{ExprCont}\texttt{]} \mid \sqsubset \cdot \sqsupset \\
\mathit{ExprCont} & ::= & \texttt{new}\ \mathit{VarType}\ \texttt{[}\mathit{Expr}\texttt{]}_1\ \ldots\ \texttt{[}\mathit{ExprCont}\texttt{]}_k \ldots \texttt{[}\mathit{Expr}\texttt{]}_n \\
 & \mid & \mathit{ExprCont}.\texttt{[}\mathit{ArgType}\texttt{]MethName (}\mathit{Expr}_1\texttt{,} \ldots \mathit{Expr}_n\texttt{)} \\
 & \mid & \mathit{Expr}.\texttt{[}\mathit{ArgType}\texttt{]MethName (}\mathit{Expr}_1\texttt{,} \ldots \mathit{ExprCont}_k\texttt{,} \ldots \mathit{Expr}_n\texttt{)} \\
 & & \quad \text{where}\ \ n \geq 1, 1 \leq k \leq n \\
 & \mid & \sqsubset \cdot \sqsupset \\
\mathit{StmtCont} & ::= & \mathit{VarCont}\ \texttt{:=}\ \mathit{Expr} \mid \mathit{Var}\ \texttt{:=}\ \mathit{ExprCont} \\
 & \mid & \texttt{if}\ \ \mathit{ExprCont}\ \ \texttt{then}\ \ \mathit{Stmts}\ \ \texttt{else}\ \ \mathit{Stmts} \\
 & \mid & \mathit{StmtCont}\ \texttt{;}\ \mathit{Stmt} \mid \texttt{return}\ \ \mathit{ExprCont} \mid \texttt{throw}\ \ \mathit{ExprCont} \\
 & \mid & \sqsubset \cdot \sqsupset
\end{array}
$$

Fig. 21. Java$_r$ exception contexts

Lemma 7 *For any Java$_r$ term* $\texttt{t}$*:* $\Gamma, \sigma \vdash_r \texttt{t} : \texttt{E-Thrn} \implies$
$\exists$ *context* $\texttt{t}' \sqsubset \cdot \sqsupset$*, and reference* $\iota_\texttt{i}$*:* $\texttt{t} = \texttt{t}' \sqsubset \texttt{throw}\ \iota_\texttt{i} \sqsupset$*, and* $\sigma(\iota_\texttt{i}) = \ll \ldots \gg^\texttt{E}$.

10 Soundness of the Java$_s$ type system

10.1 Conforming environments and states

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. Furthermore, an environment conforms to another environment if it contains all definitions from the latter, plus possibly some additional variable definitions.

Definition 13 *A value* $\texttt{val}$ weakly conforms to a type $\texttt{T}$ *in an environment* Γ *and a state* σ *iff:*

- $\texttt{val}$ *is a primitive value,* $\texttt{T}$ *is a primitive type, and* $\texttt{val} \in \texttt{T}$*, or*
- $\texttt{val} = \texttt{null}$*, and* $\texttt{T}$ *is a class, interface or array type, or*
- $\texttt{val} = \iota_\texttt{j}$*,* $\sigma(\iota_\texttt{j}) = \ll \ldots \gg^\texttt{C}$*, and* $\Gamma \vdash \texttt{C} \leq_{wdn} \texttt{T}$*, or*
- $\texttt{val} = \iota_\texttt{j}$*,* $\sigma(\iota_\texttt{j}) = [\![\ldots]\!]^{\texttt{T}'\texttt{[]}_1 \ldots \texttt{[]}_\texttt{k}}$ *and* $\Gamma \vdash \texttt{T}'\texttt{[]}_1 \ldots \texttt{[]}_\texttt{k} \leq_{wdn} \texttt{T}$.

A value $\texttt{val}$ conforms to a type $\texttt{T}$ *in an environment* Γ *and a state* σ *iff* $\texttt{val}$ *weakly conforms to* $\texttt{T}$ *in* Γ *and* σ *and*

- $\texttt{val} = \iota_\texttt{j}$*,* $\sigma(\iota_\texttt{j}) = \ll \texttt{v}_1\ \texttt{C}_1 : \texttt{val}_1, \ldots \texttt{v}_\texttt{n}\ \texttt{C}_\texttt{n} : \texttt{val}_\texttt{n} \gg^\texttt{C}$*, and* $\forall$ *labels* $\texttt{v}$*, classes* $\texttt{C}'$*, types* $\texttt{T}'$ *with* $(\texttt{C}', \texttt{T}') \in \mathit{FDecs}(\Gamma, \texttt{C}, \texttt{v})$*,* $\exists \texttt{k} \in \{\texttt{1} \ldots \texttt{n}\}$ *with* $\texttt{v}_\texttt{k} = \texttt{v}$*,* $\texttt{C}_\texttt{k} = \texttt{C}'$*, and* $\texttt{val}_\texttt{k}$ *weakly conforms to* $\texttt{T}'$ *in* Γ *and* σ*; or*

– $\mathtt{val}{=}\iota_{\mathtt{j}}$, $\sigma(\iota_{\mathtt{j}}) = [\![\mathtt{val}_0, ...\mathtt{val}_{\mathtt{n}}]\!]^{\mathtt{T}'[]_1...[]_{\mathtt{k}}}$, *and* $\forall \mathtt{i} \in \{\mathtt{0}...\mathtt{n}\} : \mathtt{val}_{\mathtt{i}}$ *weakly conforms to* $\mathtt{T}'[]_2...[]_k$.

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,σ;

– *if* $\sigma(\iota_{\mathtt{i}}) = \ll ... \gg^{\mathtt{C}}$, *then* $\iota_{\mathtt{i}}$ *conforms to* $\mathtt{C}$ *in* Γ, σ;

– *if* $\sigma(\iota_{\mathtt{i}}) = [\![...]\!]^{\mathtt{T}[]_1...[]_{\mathtt{n}}}$, *then* $\iota_{\mathtt{i}}$ *conforms to* $\mathtt{T}[]_1...[]_{\mathtt{n}}$ *in* Γ, σ.

Finally, an environment Γ conforms to environment Γ' *iff for any identifier* $\mathtt{x}$:

– $\Gamma'(\mathtt{x}) \neq \mathtt{Undef}$ *implies* $\Gamma(\mathtt{x}) = \Gamma'(\mathtt{x})$;

– $\Gamma'(\mathtt{x}) = \mathtt{Undef} \neq \Gamma(\mathtt{x})$, *implies that* $\Gamma(\mathtt{x})$ *is a variable.*

For example, the state σ_0 from section 8 conforms to the environment Γ_0 from section 3.3. The "fitting" requirement for array types from definition 12 is weaker than conforming. Also, conforming is defined in terms of an environment, whereas fitting is defined in terms of the more restricted information that is available in the program.

The following lemma states that conforming environments preserve all properties.

Lemma 8 *Given environments* Γ, Γ', *where* Γ *conforms to* Γ', *any term* $\mathtt{t}$, *types* $\mathtt{T}$, $\mathtt{T}'$ *program* $\mathtt{p}$, *and argument types* $\mathtt{AT} = \mathtt{T}_2 \times ... \times \mathtt{T}_{\mathtt{n}}$:

– $\Gamma \vdash \diamond \implies \Gamma' \vdash \diamond$;

– $\Gamma' \vdash \mathtt{p}\ \diamond \implies \Gamma \vdash \mathtt{p}\ \diamond$;

– $\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}' \iff \Gamma' \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'$;

– $\Gamma' \vdash \mathtt{t} : \mathtt{T} \implies \Gamma \vdash \mathtt{t} : \mathtt{T}$;

– $\mathit{FirstFit}(\Gamma, \mathtt{m}, \mathtt{T}', \mathtt{AT}) = \mathit{FirstFit}(\Gamma', \mathtt{m}, \mathtt{T}', \mathtt{AT})$;

– $\Gamma' \vdash_{se} \mathtt{t} : \mathtt{T} \implies \Gamma \vdash_{se} \mathtt{t} : \mathtt{T}$;

– $\Gamma', \sigma \vdash_r \mathtt{t} : \mathtt{T} \implies \Gamma, \sigma \vdash_r \mathtt{t} : \mathtt{T}$.

10.2 Properties of term evaluation

The operational semantics is deterministic up to renaming of identifiers and addresses. A term containing an actually thrown exception not included by a `try` statement, *i.e.* one with the type `E-Thrn`, will either not terminate, or it will terminate in a `throw` statement. Rewriting variables on the left hand side of assignments does not make their type more special, except for arrays. Program execution may modify the contents of arrays and objects, but will not change their type or class.

Lemma 9 *For a state σ conforming to a well-formed environment Γ, a Java$_{se}$ program with $\Gamma \vdash \mathtt{p}\ \diamond$, a well-typed Java$_r$ term* $\mathtt{t}$:

- $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{t}', \sigma' \rangle$ *and* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{t}'', \sigma'' \rangle$ *implies that* $\mathtt{t}' = \mathtt{t}''$, $\sigma' = \sigma''$ *up to renaming of addresses and identifiers. Also,* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \sigma' \rangle$ *and* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \sigma'' \rangle$ *implies that $\sigma' = \sigma''$ up to renaming of addresses and identifiers. Furthermore, it is impossible to have* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{t}'', \sigma'' \rangle$ and $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \sigma' \rangle$.
- *If* $\Gamma, \sigma \vdash_r \mathtt{t} : \mathtt{E\text{-}Thrn}$, *then, either* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}}{}^*$ *does not terminate, or* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}}{}^* \langle \mathtt{throw}\ \iota_{\mathtt{i}}, \sigma \rangle$
- *For Java$_r$ variables* $\mathtt{v}$, $\mathtt{v}'$, *if* $\langle \mathtt{v}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{v}', \sigma' \rangle$, *and* $\Gamma, \sigma \vdash_r \mathtt{v} : \mathtt{T}$, *and* $\mathtt{v}$ *is not l-ground, then* $\Gamma, \sigma \vdash_r \mathtt{v}' : \mathtt{T}'$, $\Gamma \vdash \mathtt{T}' \leq_{wdn} \mathtt{T}$ *and* $\mathtt{v}'$ *is not ground. Furthermore, if* $\mathtt{v}$ *is not an array access, then* $\mathtt{T} = \mathtt{T}'$.
- *If* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{t}', \sigma' \rangle$, *then for any* $\iota_{\mathtt{i}}$, *if* $\sigma(\iota_{\mathtt{i}}) = [\![\ldots]\!]^{\mathtt{T}[]_1 \ldots []_n}$ *then* $\sigma'(\iota_{\mathtt{i}}) = [\![\ldots]\!]^{\mathtt{T}[]_1 \ldots []_n}$, *and if* $\sigma(\iota_{\mathtt{i}}) = \ll \ldots \gg^{\mathtt{C}}$ *then* $\sigma'(\iota_{\mathtt{i}}) = \ll \ldots \gg^{\mathtt{C}}$.

Don Syme pointed out to us [27, 28] that a lemma stating that program execution preserves types up to widening, is necessary for the proof of subject reduction. Interestingly, it turned out that a stronger lemma, than that originally suggested and used in the subject reduction theorem is possible. Namely, execution of a program does not have *any* effect on the type of an expression. This lemma is easier to prove, and considerably facilitates the proof of subject reduction.

Lemma 10 *For Java$_r$ terms* $\mathtt{t}$, $\mathtt{t}'$, $\mathtt{t}''$, *states* σ, σ', *environments* Γ, Γ', *type* $\mathtt{T}''$, *Java$_s$ program* $\mathtt{p}$ *and Java$_r$ program* $\mathtt{p}' = \mathcal{C}\{\!\{\Gamma, \mathtt{p}\}\!\}$, *if*

- $\Gamma \vdash \mathtt{p}\ \diamond$ *and* $\Gamma, \sigma \vdash_r \mathtt{t} : \mathtt{T}$ *and* $\Gamma, \sigma \vdash_r \mathtt{t}'' : \mathtt{T}''$;
- σ *conforms to* Γ *and* Γ' *conforms to* Γ
- $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}'} \langle \mathtt{t}', \sigma' \rangle$, *or* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}'} \langle \sigma' \rangle$

then

- $\Gamma', \sigma' \vdash_r \mathtt{t}'' : \mathtt{T}''$'.

The lemma may be surprising: As stated later in the subject reduction theorem, a term $\mathtt{t}$ when rewritten to a new term $\mathtt{t}'$ has, possibly, a *narrower* type; therefore, one would expect evaluation of the term $\mathtt{t}$ to affect the type of a third term $\mathtt{t}''$. However, according to the above lemma, even if $\mathtt{t}''$ should contain $\mathtt{t}$ as a subterm, its type does not change. The lemma is proven by structural induction over term execution (*i.e.* on $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}'} \langle \mathtt{t}', \sigma' \rangle$, or $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}'} \langle \sigma' \rangle$), and then, each case by structural induction on the typing of $\mathtt{t}''$ (*i.e.* on $\Gamma, \sigma \vdash_r \mathtt{t}'' : \mathtt{T}''$). The interesting cases are those where the state changes directly, *i.e.* the application of the different assignment rules from figure 17, object and array creation in figure 15, and method call in figure 18. Object or array creation and method call

only introduce new entities into the state, and so not affect the type of term $\mathtt{t}''$. Assignments do not change the types of variables (these are looked up in the environment). They do not change the type of addresses (as shown in lemma 9). They do not change the type of array access because this depends on the type of the array and not on the type of the actual array component. And they do not change the type of object access because this too depends on the type of the object and the class stored in the descriptor and not on the value stored in the object field.

The *array property*, introduced in the following definition, ensures that checking for fitting when executing array assignments will be sufficient to preserve conformance of the state.

Definition 14 *A Java$_r$ term* $\mathtt{t}$ *has the* array property *for a program* $\mathtt{p}$ *and for a state* σ*, iff for any subterm of* $\mathtt{t}$ *with the form* $\mathtt{v[e]} := \mathtt{e}'$*, with* $\Gamma, \sigma \vdash_r \mathtt{v[e]} : \mathtt{T}$ *and* $\Gamma, \sigma \vdash_r \mathtt{e}' : \mathtt{T}'$*, if NOT* $\Gamma \vdash \mathtt{T}' \leq_{wdn} \mathtt{T}$*, then for appropriate* $\mathtt{n} \geq 0$*,* $\mathtt{T} = \mathtt{C}[]_1 \ldots []_n$*,* $\mathtt{T}' = \mathtt{C}'[]_1 \ldots []_n$*, and NOT* $\mathtt{p} \vdash \mathtt{C}' \sqsubseteq \mathtt{C}$*.*

The array property is trivially guaranteed in type-correct Java$_{se}$ terms, and thus also in any Java$_r$ terms that are the result of enriching type-correct Java$_s$ terms, and it is preserved by the execution of Java$_r$ terms.

Lemma 11 *For an environment* Γ *under which the Java$_s$ term* $\mathtt{t}$ *and Java$_{se}$ term* $\mathtt{t}'$ *are well typed, Java$_{se}$ program* $\mathtt{p}$ *with* $\Gamma \vdash \mathtt{p} \diamond$*,* $\mathtt{p}' = \mathcal{C}\{\!\{\Gamma, \mathtt{p}\}\!\}$ *:*

- $\mathtt{t}'$ *has the array property for* $\mathtt{p}$ *and any state* σ*.*
- $\mathcal{C}\{\!\{\Gamma, \mathtt{t}\}\!\}$ *has the array property for* $\mathtt{p}$ *and any state* σ*.*
- *If* σ *conforms to* Γ*,* $\mathtt{t}'$ *has the array property for* $\mathtt{p}'$*,* σ*, and* $\langle \mathtt{t}', \sigma \rangle \rightsquigarrow_{\mathtt{p}'} \langle \mathtt{t}'', \sigma' \rangle$*, then* $\mathtt{t}''$ *has the array property for* $\mathtt{p}'$ *and* σ'*.*
- $\forall$ *Java$_r$ terms* $\mathtt{t}''$*, states* σ *:* $\mathtt{t}''$ *has the array property for* $\mathtt{p}$ *and* $\sigma \implies$ $\mathtt{t}''$ *has the array property for* $\mathtt{p}'$ *and* σ

10.3 Subject reduction and soundness

The subject reduction theorem says that any non-ground well-typed Java$_{se}$ term either rewrites to another well-typed term of a type that can be widened to the type of the original term, or it rewrites to an exception. Furthermore, the state remains consistent with the environment. The subject reduction theorem of this paper is stronger than usual subject reduction theorems: not only does it guarantee that rewriting preserves types, but it also guarantees that a rewrite step exists for any well-formed, non-ground term. (In that sense it combines the safety and soundness property described in chapter 4 of this book.) In particular, it guarantees for statically type-correct expressions, that the situation where an object cannot execute a message (the Smalltalk counterpart to "`object does not understand message`"[14]) will never occur. On the other hand, it does not preclude the usual run-time errors like index out of bound, or wrong assignment

to array components; however, it *does* guarantee that such erroneous situations will raise an exception, as opposed to going unnoticed and corrupting the runtime environment.

Theorem 1 Subject Reduction *For a state σ that conforms to an environment Γ, a Java$_{se}$ program* p *with* $\Gamma \vdash_{se}$ p $\gg$, *a non-ground Java$_r$ expression* t *with the array property for* p *and σ, and a type* T *with* $\Gamma, \sigma \vdash_r$ t : T, *there exist σ', Γ',* t′, T′ *such that:*

- $\langle$t, $\sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle$t′, $\sigma'\rangle$, *and* $\Gamma', \sigma' \vdash_r$ t′ : T′, *and* t′ *has the array property for* p *and σ', and:*
  - T′=E-Thrn, E *an exception, σ' conforms to Γ,* $\Gamma' = \Gamma$ *or*
  - $\Gamma \vdash$ T′ $\leq_{wdn}$ T, Γ' *conforms to Γ, σ' conforms to Γ'*

  or
- $\langle$t, $\sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle\sigma'\rangle$ *and* σ' *conforms to* Γ

Furthermore, if t *is a non l-ground variable, then* $\langle$t, $\sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle$t′, $\sigma'\rangle$ *and* t′ *is not ground. Also, if* t *is a non l-ground variable which isn't an array access, then* T = T′.
The theorem is proven by structural induction over the derivation of $\Gamma, \sigma \vdash_r$ t : T.

When the method call aPhil.[Phil]think(aPhil) was evaluated in the philosophers example, after the third rewrite step, the "environment extension" required by the subject reduction theorem is $\Gamma' = \Gamma_0$, w : FrPhil, w′ : FrPhil. The states σ_1, σ_2 conform to Γ'.

Finally, the soundness theorem states that execution of a well-typed Java$_s$ program will produce a uniquely defined value of the expected type in a state conforming to the definitions, or it will throw an exception which will be propagated to the outermost level, or it will not terminate.

Theorem 2 Soundness *Take any Java$_s$ expression* t, *a well-formed environment Γ, a type* T *with* $\Gamma \vdash$ t : T, *a Java$_s$ program* p *with* $\Gamma \vdash$ p $\gg$, *and a state σ that conforms to Γ. Then for the Java$_{se}$ program* p′, p′ $= \mathcal{C}\{\!\{\Gamma,$ p$\}\!\}$, *there exists a unique Java$_r$ expression* t′, *and a state σ', such that:*

- T $\neq$ void, $\langle \mathcal{C}\{\!\{\Gamma,$ t$\}\!\}, \sigma\rangle \rightsquigarrow_{\mathtt{p}'}{}^{*} \langle$t′, $\sigma'\rangle$, t′ *is ground,* $\exists$T′ : $\Gamma, \sigma' \vdash_r$ t′ : T′, $\Gamma \vdash$ T′ $\leq_{wdn}$ T *and σ' conforms to Γ or*
- T = void, *and* $\langle \mathcal{C}\{\!\{\Gamma,$ t$\}\!\}, \sigma\rangle \rightsquigarrow_{\mathtt{p}'}{}^{*} \langle\sigma'\rangle$ *and σ' conforms to Γ or*
- $\langle \mathcal{C}\{\!\{\Gamma,$ t$\}\!\}, \sigma\rangle \rightsquigarrow_{\mathtt{p}'}{}^{*}$ *does not terminate or*
- $\langle \mathcal{C}\{\!\{\Gamma,$ t$\}\!\}, \sigma\rangle \rightsquigarrow_{\mathtt{p}'}{}^{*} \langle$throw $\iota_{\mathtt{i}}, \sigma'\rangle$, *and* $\sigma(\iota_{\mathtt{i}}) = \ll\ldots\gg^{\mathtt{E}}$, *and* $\Gamma \vdash$ E $\sqsubseteq$ Exception

11 Conclusions

We have given a formal description of the operational semantics and type system for a substantial subset of Java. We believe this subset is reasonably rich and contains 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 think, does not diminish the application of our results.

Close scrutiny of the language description showed that the semantic issues related to the scope of our investigation are unambiguously answered by [16]. However, we found areas that could have been defined more generally (*e.g.* the return types of methods override those from superclasses and superinterfaces) and others that could have been defined more concisely (*e.g.* the descriptions of widening and of exceptions). Furthermore, in [21, 31] we describe problems related to the definition of binary compatibility, and attempt a formalization of this concept.

We believe that the formal system we have developed is very near to Java and to programmers' intuitive ideas about program execution. On the other hand, we now have a large system, and the proofs of the lemmas require the consideration of many cases. The system grew and evolved through many iterations, and during which some omissions crept into the argumentation. The most significant omissions were uncovered by Don Syme and are described earlier on in this chapter, and also, in chapter 5 of this book. With the modifications he suggested, he was able to validate the subject reduction theorem using his theorem checker. This gives us greater confidence in our results, but it also underlines the importance of the use of theorem checkers for such, rather large systems.

Another proof of the soundness of the Java type system, using Isabelle in a large step semantics is described in [23] and in the next chapter of this book. Applications of theorem provers for programming language properties are also described in [15, 26, 30].

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 its possible extensions [24, 3, 2]. We are also looking at further language properties such as an abstraction property and binary compatibility [31].

Acknowledgments

We would like to acknowledge encouragement from our colleagues in the Department of Computing during the formulation of these ideas and financial support from the EPSRC (Grant Refs:GR/L 76709 and GR/K 73282).

We are greatly indebted to many people for valuable feedback: Peter Sellinger, David von Oheimb, Yao Feng, Gabrielle Sinnadurai, David Wragg, Guiseppe Castagna, Sarfraz Khurshid, the TAPOS referees, the referees of this book, and most particularly to Don Syme.

References

1. M. Abadi and L. Cardelli. A Semantics of Object Types. In *LICS'94 Proceedings*, 1994.
2. Ole Ageson, Stephen Freunds, and John C. Mitchell. Adding Parameterization to Java. In *OOPSLA'97 Proceedings*, 1997.
3. Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized Types and Java. In *POPL'97 Proceedings*, January 1997.
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 *OOPLSA'89 Proceedings*, pages 457–467, 1989.
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. Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? In *Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997.
12. Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In *Proceedings of the European Conference on Object-Oriented Programming*, June 1997.
13. Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is Java Sound? *Theory and Practice of Object Systems*, 1998. to appear, available at http://www-dse.doc.ic.ac.uk/projects/slurp/.
14. A. Goldberg and D. Robson. *SmallTalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.
15. M. Gordon and T.F. Melhams, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
16. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, August 1996.
17. R. Harper. A simplified account of polymorphic references. Technical Report CMU-CS-93-169, Carnegie Mellon University, 1993.
18. Daniel Ingalls. The Smalltalk-76 programming system design and implementation. In *POPL'78 Proceedings*, pages 9–15, January 1978.
19. The Java Language Specification, May 1996.
20. John Boyland and Giuseppe Castagna. Parasitic Methods: Implementation of Multimethods for Java. Technical report, C.N.R.S, November 1997.
21. Sarfraz Khurshid. Some Aspects of Type Soundness for Java, 1997. BSc thesis.
22. Bertrand Meyer. Static typing and other mysteries of life, http://www.eiffel.com 1995.

23. Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe — definitely. Technical report, Technische Universitaet Muenchen, 1997. Submitted for publication.
24. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *POPL'97 Proceedings*, January 1997.
25. Peter Sellinger. private communication, October 1996.
26. Donald Syme. DECLARE: A Prototype Declarative Proof System for Higher Order Logic. Technical Report 416, Cambridge University, March 1997.
27. Donald Syme. Private Communication, 1997.
28. Donald Syme. Proving Java Type Sound. Technical Report 427, Cambridge University, June 1997.
29. Mads Tofte. Type Inference for Polymorphic References. In *Information and Computation'80 Conference Proceedings*, pages 1–34, November 1980.
30. Myra VanInwegen. Towards Type Preservation in Core SML. Technical report, Cambridge University, 1997.
31. David Wragg, Sophia Drossopoulou, and Susan Eisenbach. Java Binary Compatibility is Almost Correct. Technical Report 3/98, Imperial College, 1998. http://www-dse/projects/SLURP/bc.
32. Andrew Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1), 1994.