# Java Type Soundness Revisited

Sophia Drossopoulou, Tanya Valkevych, Susan Eisenbach  
Department of Computing  
Imperial College of Science, Technology and Medicine  
email: {sd,tanya,se}@doc.ic.ac.uk  
September 8, 2000

## Contents

1. **Introduction**  
2. **Examples in Java**  
   2.1 Philosophers, an example demonstrating inheritance and overloading  
   2.2 Clubs, an example demonstrating arrays  
   2.3 Worries, an example demonstrating exceptions  
   2.4 Students, an example demonstrating separate compilation and linking  
3. **Our approach and its rationale**  
   3.1 Java³, Java², Java¹  
   3.2 The run-time model  
   3.3 Run-time exceptions  
   3.4 Descriptions, typing, compilation, and link-time checks  
   3.5 Normal and abnormal types  
   3.6 Widening and subject reduction  
4. **The language Java²**  
5. **Type notions, descriptions, and environments**  
   5.1 Type notions  
   5.2 Descriptions  
   5.3 Unambiguous descriptions  
   5.4 Acyclic descriptions, look-up, widening  
   5.5 Properties of acyclic descriptions  
   5.6 Well-formed and complete descriptions, and their properties  
   5.7 Environments  
6. **The Java² type system**  
   6.1 Normal types for Java² terms  
   6.2 Abnormal types for Java² terms  
   6.3 Properties of the Java² type system  
   6.4 Well-formed Java² programs  
7. **Java³, enriching Java²**  
8. **Compilation of Java³ programs**
9 The Java\(^b\) type system  
  9.1 Properties of the Java\(^b\) type system  
  9.2 Well-formed and complete Java\(^b\) programs

10 Java\(^r\), the run-time language

11 The operational semantics  
  11.1 State and object operations, ground terms  
  11.2 Execution

12 The Java\(^r\) type system  
  12.1 Properties of the Java\(^r\) type system

13 Soundness of the Java type system  
  13.1 Conformance  
  13.2 Properties of term evaluation  
  13.3 Subject reduction and soundness

14 Java as a fragment system  
  14.1 Fragment systems  
  14.2 Elements of a Java fragment system  
  14.3 Properties of Java\(^r\), Java\(^b\) programs

15 Conclusions
Abstract

We present an operational semantics, type system, and a proof of type soundness for a substantial subset of Java. The subset includes interfaces, classes, inheritance, field hiding, method overloading and overriding, arrays with associated dynamic checks, and exception handling.

We distinguish between normal execution, where no exception is thrown – or, more precisely, any exception thrown is handled – and abnormal execution, where an exception is thrown and not handled. The type system distinguishes normal types which describe the possible outcomes of normal execution, and abnormal types which describe the possible outcomes of abnormal execution. The type of a term consists of its normal type and its abnormal type.

With this set-up we prove subject reduction. Thus, the meaning of our subject reduction theorem is stronger than usual: it guarantees that normal execution returns a value of a type compatible with the normal type of the term, and that abnormal execution throws an exception compatible with the abnormal type of the term.

We also give a formalization of separate compilation and linking. We distinguish between Java^r, Java^w and Java^s, which stand for the source language, the binary language and the run-time language. Java^r closely reflects the Java syntax. Java^w is a version of Java where field access and method call are enriched with type information; we consider Java^s a high-level version of bytecode. Java^w is an extension of Java^r where we allow for addresses. Compilation maps Java^r onto Java^w and uses type information from Java^s for the imported classes. The linker checks are represented as well-formedness checks for Java^w.

Finally, we show that Java^r and Java^w can be viewed as a fragment system – an abstract model of separate compilation and linking which was introduced to investigate possible meanings of binary compatibility.

1 Introduction

Java offers a new, revolutionary paradigm for distribution, maintenance and execution of programs. With the advent of the Internet, Java has moved into a position no other programming language has ever been in before. Millions of naive computer users download applets and execute them, mostly unaware of their usage of Java. The implications on security and integrity are tremendous. It has been demonstrated that many breaches of the Java security originate with the possibility of breaking the type system through a combination of fooling the bytecode verifier and the linker/loader [4, 5, 23].

The semantics of the Java source language [7, 35, 37, 27, 1, 29, 8], the Java bytecode [33, 12, 28, 26, 29], and safety pitfalls or security considerations [4, 24, 5, 38, 23] has attracted much research. In some cases, this research has helped to get some loopholes fixed [32]; in other cases, it has illuminated grey areas and explored possible interpretations [12, 35, 9, 33, 11].

This paper presents a further stage in our ongoing research into the semantics of the Java language. Java was designed with the aim to be conservative in its inclusion of new language features [15]. Nevertheless, a formalization is worthwhile. Several of these features had not been formalized before, and the interplay of features always opens possibilities for unsound systems. The Java features we found most interesting in terms of their modeling are:

- run-time aliasing,
- method overloading combined with method overriding,
- field hiding,
- exception throwing, handling and the associated throws clauses,
- arrays with covariant subtyping and run-time checks,
- separate compilation and link-time checks.
We have also worked on a formalization of constructors and private/public modifiers [18], the meaning of binary compatibility [11] and on a formalization of dynamic linking and binary compatibility [6].

Our approach is an extension to earlier work [7, 8]. We define Java\(^a\), a provably safe subset of Java containing some primitive types, interfaces, classes with instance variables and instance methods, inheritance, hiding of instance variables, overloading and overriding of instance methods, arrays, implicit pointers and the null value, object creation, assignment, field and array access, method call and dynamic method binding, exceptions and exception handling.

\[
\text{Java} \supset \text{Java}^a \longrightarrow \text{Java}^b \subset \text{Java}^c \leftarrow \text{Java}^d
\]

\[
\text{Type} = \text{Type} \supset \text{Type} \supset \text{Type} \supset \text{Type}
\]

The execution of Java programs requires some type information at run-time (e.g. field and method descriptors). For this reason, we define Java\(^b\), an enriched version of Java\(^a\) containing compile-time type information necessary for the execution of method call and field access. Java\(^b\) corresponds to a high-level representation of the bytecode [21]. Compilation, \( \mathcal{C} \), maps Java\(^a\) terms to Java\(^b\) terms preserving their types. Execution of run-time terms may produce terms which are not described by Java\(^b\). We therefore extend Java\(^b\), obtaining Java\(^c\), which describes run-time terms, including references.

Execution may proceed normally, where no exception is thrown – or, more precisely, any exception thrown is handled – or abnormally, where an exception is thrown and not handled. A term therefore has a type which is a pair of a normal type and an abnormal type. The normal type describes the possible outcomes of normal execution. A normal type may be either a class, or an interface, or a primitive type, like int, char, boolean, nil or void, or \( \bot \). The least normal type \( \bot \) describes terms which definitely will throw exceptions. The abnormal type describes the possible outcomes of abnormal execution. An abnormal type is represented by a set of subclasses of the predefined Throwable class.

Excluding \( \bot \), normal types are “usual” types, whereas abnormal types correspond to effect systems [22, 36]. Rewriting a term preserves both the normal type and the abnormal type; any exception it may raise is described by the abnormal type, i.e. by the effect system.

We define widening for types. A type \( T \) widens to another type \( T' \), if the normal part of \( T \) is \( \bot \), or identical to or a subclass of, or subinterface of, or a subclass of class that implements an interface of, the normal part of \( T' \), and if the abnormal part of \( T \) consists of classes which are unchecked exceptions, or identical to or subclasses of, corresponding ones in the abnormal part of \( T' \).

We prove a subject reduction theorem [40] which states that term execution preserves types up to widening. Our definition of types as pairs of normal—abnormal types makes the meaning of the subject reduction theorem stronger, namely that normal execution will return a value of a type compatible with the normal type, and abnormal execution will throw an exception whose class is a subclass of one of the classes in its abnormal type.

For our treatment of separate compilation and linking, we consider Java\(^b\) to stand for a high-level view of the bytecode, the mapping \( \mathcal{C} \) for a formalization of compilation, and the Java\(^b\) well-formedness for the linker checks. \( \mathcal{C} \) maps Java\(^a\) onto Java\(^b\) (Java compilers create *.class files out of *.java files), and uses the type information from Java\(^a\) for the imported classes (Java compilers read *.class files). \( \mathcal{C} \) may produce a Java\(^b\) program even if the imported *.class files reference to further classes whose *.class files are absent). Java\(^b\) checks check Java\(^b\) programs using further Java\(^b\) programs (Java linkers check bytecode against further bytecode) and ensure that all the external references of a Java\(^b\) class/interface are satisfied by the context Java\(^b\) classes/interfaces. It is possible for a Java\(^b\) class/interface to be well-formed in the context of Java\(^b\) classes/interfaces which are not well-formed (it is possible for bytecode to link successfully into code which itself will not link successfully when its references need resolving). So far, our description is faithful to
Java. However, we have proven subject reduction only for an eagerly linked program, where all
external references have been resolved, before execution. This is a point we plan to tackle in the
future.

Finally, in [9], we introduced fragment systems, to reason about separate compilation and
linking at a more abstract, language independent level, and to explore the possible meanings of
binary compatibility and their properties. In this paper we demonstrate how Java⁶ and Java⁷ can
be understood to form a fragment system.

The novelty of this work compared to the earlier ones [7, 8] consists in

- the incorporation of the abnormal type which guarantees that any exception thrown was
  predicted in the throws clause of the method being executed,
- the description of separate compilation,
- the description of Java as a fragment system [9],
- the adoption of a syntax for Java⁶ which is near to the Java actual syntax.

The rest of this paper is organized as follows. In section 2 we give examples in Java which
illustrate the concepts modeled in this work. In section 3 we explain our approach using these
examples.

In section 4 we give the syntax of Java⁶. Type notions, descriptions, and environments are
defined in section 5. In section 6 we define types for Java⁶ and describe well-formed Java⁶ programs.

In section 7 we define the language Java⁶. Compilation from Java⁶ to Java⁷ is described in
section 8. In section 9 we give types to Java⁷ terms and describe well-formed and complete Java⁷
programs.

In section 10 we define the run-time language Java⁷. States, configurations and the operational
semantics for Java⁷ are described in section 11. In section 12 we give types to Java⁷ terms. In
section 13 we state properties of the operational semantics and, in particular, the subject reduction
theorem.

In section 14, we study properties of programs consisting of several components, and
demonstrate how Java⁶ and Java⁷ can be understood to form a fragment system as defined in [9]. Finally,
in section 15 we draw some conclusions and describe further work. We also provide an index with
classified entries for terminology, judgements, functions, identifiers, Java features and examples
from this paper.

2 Examples in Java

We give four examples in Java which illustrate the most interesting aspects of the subset we discuss,
namely, inheritance and overloading, arrays, exceptions, separate compilation and linking. These
examples will also be used through the text as running examples.

2.1 Philosophers, an example demonstrating inheritance and overloading

The following, admittedly contrived, example $P_{ph}$ serves to demonstrate issues around inheritance,
instance variable hiding, method overriding and method overloading.

$P_{ph}$ can have the following interpretation. Philosophers like the Truth. When a philosopher
thinks about some question (class Quest), he produces a Book. Besides, when a philosopher
thinks about existentialism (class Exist), he refers to a French philosopher. French philosophers
like Food. When they think about questions, they also produce a Book.

Assuming previous definitions of classes Book, Food, Truth and variable oyster of type Food,
consider the classes Quest, Exist, Phil, FrPhil, Test which are defined in the program $P_{ph}$ as:
```java
P^p_h =

class Quest {}
class Exist extends Quest {}

class Phil {
    Truth like;
    Book think(Quest y) { /* thinkBody1 */ ... }
    FrPhil think(Exist y) { /* thinkBody2 */ ... }
}
class FrPhil extends Phil {
    Food like;
    Book think(Quest y) { this.like == oyster; ... }
}
class Test {
    void test(Quest aQuest, Exist being, Phil aPhil, FrPhil pascal) {
        aPhil.think(aQuest); // executes thinkBody1
        aPhil.think(being); // executes thinkBody2
        pascal.think(aQuest); // executes this.like == oyster; ...
        pascal.think(being); // ambiguous
        ... aPhil.like ... // returns Truth
        ... pascal.like ... // returns Food
        aPhil = pascal;
        ... aPhil.like ... // returns Truth
        aPhil.think(aQuest); // executes this.like == oyster; ...
        aPhil.think(being); // executes thinkBody2
    }
}
```

The above example demonstrates:

- Scopes are recursive, e.g. the class FrPhil is visible inside the class Phil, i.e. before its declaration.

- Instance methods can be overloaded, i.e. it is valid to have more than one method with the same name but different arguments in a class. So, there are two methods think in the class Phil: one with argument type Quest, the other with argument type Exist.

- Classes inherit methods from their superclasses. Thus, class FrPhil inherits method FrPhil think(Exist y) { /* thinkBody2 */ ... } from class Phil.

- An instance method overrides an inherited instance method with the same identifier and argument types. Therefore, the method Book think(Quest y) { this.like == oyster; ... } in class FrPhil overrides the method Book think(Quest y) { /* thinkBody2 */ ... } from class Phil.

- Instance variables hide instance variables with a same identifier declared in superclasses. Thus, Food like in class FrPhil hides the instance variable declaration Truth like in class Phil. This feature allows for static types in field access, 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.

- Methods are bound according to the dynamic class of the receiver and the static type of the arguments. So, the call aPhil.think(aQuest) will execute /* thinkBody1 */ ... if aPhil is an object of class Phil, and it will execute this.like == oyster; ... if aPhil is an object of class FrPhil. Because overloaded methods are bound according to the static type of the arguments, with aPhil = new Phil(); the code aQuest = new Quest();
aphil.think(aQuest) executes /* thinkBody */ ... and aQuest = new Exist();
aphil.think(aQuest) also executes /* thinkBody */ .... The call pascal.think(being) is ambiguous, because the method Book think(Quest y) from class FrPhil and the method FrPhil think(Exist y) from class Phil are both applicable, and neither is more special than the other.

2.2 Clubs, an example demonstrating arrays

The following example P_cl demonstrates issues around arrays, namely array declaration, creation and access, covariant array types, the associated run-time type checks and exceptions.

P_cl is based on the philosophers example. It can be understood as describing clubs of thinking philosophers and dining French philosophers. Initially, the membership of the thinking club is restricted to four philosophers, and the membership of the dining club is restricted to three French philosophers. Later on, the thinking club becomes a dining club, and so, only French philosophers may join.

P_cl = ...
Phil aPhil = new Phil();
Phil[] thinking;
thinking = new Phil[4];
thinking[3] = aPhil;
thinking[3] = pascal;
thinking[3] = aPhil;
dining = thinking; // type error

thinking = dining;  // covariant array subtypes
thinking[2] = aPhil; // throws ArrayStoreException
thinking[2] = pascal;
thinking[3] = pascal; // throws IndexOutOfBoundsException
...

The above example demonstrates:

- Array types consist of the element type and the number of dimensions (in empty square brackets), e.g. the declaration Phil[] thinking defines the variable thinking which will point to one-dimensional arrays of objects of class Phil or a subclass of Phil.

- Array creation specifies the element type, the numbers of dimension, and the length of the array for at least one of the dimensions, e.g. the expression new Phil[4] creates a one-dimensional array with four philosophers.

- Array component access requires a nonnegative integer index value between 0 and the length of that dimension decreased by 1. That is why
dining = new FrPhil[3]; dining[3];
throws an IndexOutOfBoundsException.

- The subtype relationship is covariant for arrays. Thus, both statements
thinking = new FrPhil[4]; thinking = dining;
are type-correct.

- Assignment of a value to an array component requires a run-time check to ensure that the value is compatible with the run-time type of the array. Therefore,
thinking = new Phil[4]; thinking[2] = new Phil();
executes successfully, whereas
thinking = new FrPhil[4]; thinking[2] = new Phil();
throws an ArrayStoreException.
2.3 Worries, an example demonstrating exceptions

Exception handling in programming languages supports the development of robust programs [13]. When an exception is raised – thrown in Java terminology – control is transferred to the nearest handler – catch clause in Java – for that exception found in the dynamic call stack.

In Java [14] exceptions are objects, and behave as such until they are thrown (i.e. they can be assigned, passed as parameters, etc.). A method must mention in the throws clause of its header all the checked exceptions that might escape from execution of its body, and the compiler ensures that callers of a method either handle this method’s potential exceptions, or they themselves explicitly mention these exceptions in their headers.

Exceptions are objects of the predefined class Throwable or its subclasses. The unchecked exceptions are exempt from the requirement of being listed in the headers. Unchecked exceptions are the subclasses of class RuntimeException describing run-time errors, e.g. dereferencing null or division by zero, and the subclasses of Error describing linkage and virtual machine errors, e.g. the absence of a *.class file or verification errors. All the other exceptions are checked exceptions.

C++ [34] and SML [25] exceptions are similar to Java exceptions in the way they are propagated and caught, and in that exceptions behave as ordinary objects/values unless explicitly thrown. Unlike Java, C++ exceptions do not need to be objects of a special exception class; SML exceptions are special values of any type which have been annotated as exception. Neither C++ nor SML programmers are expected to declare the exceptions which escape a function.\(^1\)

The following example P_wr demonstrates some of these issues.

\[P_{wr} = \]

```java
class Worry extends Exception {

    class Illness extends Exception {
        int severity;
        Illness treat() throws Worry {
            if(...) throw new Worry();
            else return this;
        }

        Illness cure() {
            severity = -10;
            return this;
        }
    }

    class Person {
        int age;
        Illness diagnose() { return new Illness(); }
        void act() throws Worry, Illness {
            try {
                if(...) throw diagnose().treat();
                else age = age+1;
            }
            catch(Illness i) { i.cure(); }
        }

        void live() throws Worry, Illness {
            act();
            Doctor d = null;
            d.act();
        }

        void study() {
            try {
                act();
            }
            catch(Throwable x) { age = age+2; }
        }
    }

    class Doctor extends Person {
        void act() throws Illness { throw diagnose(); }
        void live() { int i = diagnose().cure().severity; }
    }
}
```

Consider the following declarations and initializations:

\(^1\)This is why the SML formal description [25] does not have a concept corresponding to our abnormal types.
Person peter; peter = new Person();
Doctor david; david = new Doctor();

The above example demonstrates:

- Exception classes are those declared as subclasses of Throwable. The predefined class Exception is a subclass of Throwable, therefore the classes Worry and Illness are exception classes.

- Exceptions are thrown by throw statements, e.g. execution of david.act() throws an exception of class Illness.

- Unless thrown, objects of an exception class behave as normal objects. So, execution of david.live() will not throw any exception: the call diagnose() returns an Illness object, which executes the method cure, and then returns its field severity. Therefore, the method live in class Doctor has no throws clause.

- A method header may mention one or more exception classes in its throws clause.

- The throws clause of a method header must mention the class or superclass of any checked exception that might be thrown during evaluation of its body, and during evaluation of any overriding method. Therefore, the method header of live in class Person, which calls the method act, has to mention the exceptions Illness and Worry, because these may be thrown when executing the nested call of act, i.e. the classes Illness and Worry. Similarly, the method header for act in class Person has to mention the class Worry, because it might be thrown in its method body, and it also has to mention the exception Illness, because Illness is a possible exception of the overriding method act from class Doctor.

- Unchecked exceptions need not be mentioned in throws clauses. That is why, although execution of Doctor d=null; d.act(); inevitably throws a NullPointerException, this exception need not be mentioned in the throws clause of the header of the method live.

- An exception thrown and caught within a method need not be mentioned in the header of that method. For example, the method study of class Person does not mention any exception because any exception potentially thrown by the nested method call act() is caught by the catch clause.

- Once an exception is thrown, it is propagated through the dynamic chain of method calls, until a handler is reached. Thus, an exception of class Worry will be thrown when evaluating peter.act(). On the other hand, the call peter.study() will not terminate with a thrown exception, because any exception is caught inside the method body of study.

- Exception propagation follows the dynamic chain of method calls, and does not take overridden methods into account. Thus, david.act() will throw an exception of class Illness. This exception will not be caught in the catch clause of act in class Person; it will be propagated to the expression that contained the method call david.act().

- In general, it is unpredictable whether a term will evaluate with or without exceptions. For example, depending on the outcome of the evaluation of the condition in the method treat, the term new Illness().treat() may throw an exception of class Worry, or it may return an object of class Illness.
2.4 Students, an example demonstrating separate compilation and linking

In Java, classes can be compiled separately and linked at run-time. Usually the compiler produces bytecode [26], but may produce any other format of intermediate code provided that there is an associated interpreter and that the intermediate code contains sufficient type information (c.f. ch. 15 in [14]). The type information in the bytecode is used to check the consistency of new bytecode as it is loaded and linked. Interestingly, we may link successfully bytecode for which the corresponding source code would be type-incorrect.

We demonstrate the above issues using the following example. The phases expresses major stages in program development.

// 1st phase
P_{st} = \text{class Student \{ \text{int grade; } \}}
P_{cs} = \text{class CStudent extends Student \{ \}}
P_{lab} = \text{class Lab \{ CStudent guy; \text{void f()\{guy.grade}=100; \}} \}

// 2nd phase
P_{cs} = \text{class CStudent extends Student \{ \text{char grade; } \}}

// 3rd phase
P_{a} = \text{class Marker \{ CStudent guy; \text{void g()\{guy.grade=}'A'\}; \}} \}

The first phase consists of the programs P_{st}, P_{cs}, P_{lab} with the classes Student, CStudent, Lab. These can be compiled and linked without errors. The bytecode program consists of binaries for P_{st}, P_{cs}, P_{lab}.

In the second phase the field grade is added to the class CStudent in program P_{cs}, and the class is compiled. The sources are not consistent any more, e.g. compilation of P_{lab} would be type-incorrect. However, the binary of P_{cs} can be linked successfully with the ones for P_{st} and P_{lab} produced in the first phase. Our program consists now of binaries for P_{st}, P_{cs}, P_{lab}.

In the third phase a new class Marker is defined in the program P_{a}. This class is type-correct and can be compiled producing binaries for P_{a}. The four binaries can be linked without errors, and our program contains binaries for P_{st}, P_{lab} from the first phase, binary for P_{cs} from the second phase and binary for P_{a} from the third phase. Notice, that in this program the compiled form of the expression guy.grade in the binary P_{lab} refers to an integer, whereas the compiled form of the same expression in the binary of P_{a} of the third phase refers to a character. The two different compiled forms exist in the same binary programs at the same time, and refer to different fields of the CStudent object. They differ through the extra annotation in the bytecode, which are also described in Java\textsuperscript{b}.

A formalization of Java should therefore reflect that compilation takes place in the context of binaries and that well-formedness of source code and well-formedness of binary code are different concepts.

3 Our approach and its rationale

In this section we outline our approach in some more detail, using the examples from the previous section.

3.1 Java\textsuperscript{a}, Java\textsuperscript{b}, Java\textsuperscript{c}

Java\textsuperscript{a} corresponds to the subset of Java source which we model in this paper. The subset comprises classes, inheritance, fields, methods, arrays, exceptions and exception handling. The syntax of Java\textsuperscript{a} is almost identical to that of Java.
As we showed in the philosophers example, execution of Java terms requires some compile-time information. For example, although the expression pascal\_like always returns a value of a class Food, the term aPhil=pascal; aPhil\_like returns a value of a class Truth. Similarly, with aPhil being an object of class Phil even if a\_quest contains an object of class Exist, the method call aPhil.\_think(a\_quest) will not execute /s thinkBody2/ / ... We model this by enriching the source terms.

Thus, we obtain Java\textsuperscript{b} – an enriched version of Java\textsuperscript{a} – and define a compilation step \( \mathcal{C} \) which maps Java\textsuperscript{a} to Java\textsuperscript{b}:

\[
\mathcal{C} : \ldots \times \text{Java}^a \rightarrow \text{Java}^b
\]

In the case of field access, we add information representing the class from which the field declaration was chosen, and thus

\[
\begin{align*}
\mathcal{C}\{…, a\text{Phil}\_like\} &= a\text{Phil}\_\text{like} \\
\mathcal{C}\{…, \text{pascal}\_like\} &= \text{pascal}\_\text{like}
\end{align*}
\]

In the case of method calls, we extend the term with information about the argument type, e.g.

\[
\begin{align*}
\mathcal{C}\{…, a\text{Phil}\_\text{think(being)}\} &= \text{pascal}\_\text{like}\_\text{think(being)} \\
\mathcal{C}\{…, \text{pascal}\_\text{think(being)}\} &= a\text{Phil}\_\text{like}\_\text{think(being)}
\end{align*}
\]

Java\textsuperscript{b} is a language that describes terms as they are being executed. Because Java implicitly treats any variable of reference type (objects) as a pointer, Java\textsuperscript{b} needs to represent pointers and the capability of aliasing.

Therefore, Java\textsuperscript{b} terms may contain addresses. For example, the term aPhil\_like may be rewritten to \( \mu_i\_\text{like} \) where \( \mu_i, \mu_j, \mu_k \) etc. represent addresses. Thus, Java\textsuperscript{b} is an extension of Java\textsuperscript{a}:

\[
\text{Java}^b \subseteq \text{Java}^a
\]

and it allows addresses wherever expressions are expected.

Java\textsuperscript{a} programs are designated by S, S', S\_1, etc. Java\textsuperscript{b} programs are designated by B, B', B\_1, etc. Java programs which may belong to Java\textsuperscript{a} or to Java\textsuperscript{b} are denoted by P, P', P\_1, etc.

### 3.2 The run-time model

Evaluation of Java\textsuperscript{b} terms takes place in the context of a state and a program. Thus, we define rewriting as follows:

\( \sim \) : Java\textsuperscript{b} program \( \rightarrow \) Java\textsuperscript{b}-term \( \times \) state \( \rightarrow \) (Java\textsuperscript{b}-term \( \times \) state) \( \cup \) (state)

The states map identifiers onto primitive values or addresses, and addresses onto objects or arrays.

The objects are annotated by their classes. This is necessary, for example, for method selection when methods are bound according to the run-time class of the receiver. The objects contain the name of their fields, the name of classes which contain the field declaration and the values of these fields. For example, an object of class FrPhil can be represented in the state \( \sigma_0 \) in the following way:

\[
\begin{align*}
\sigma_0(a\text{Phil}) &= \mu_1 \\
\sigma_0(\mu_1) &= \llike \text{Phil: } \mu_2, \like \text{FrPhil: } \text{null}\rr_{\text{FrPhil}} \\
\sigma_0(\mu_2) &= \llike \text{Truth} \\
\sigma_0(\mu_3) &= \llike \text{Food}
\end{align*}
\]

The arrays are annotated by array types, which include the array element type and dimension. Arrays consist of a sequence of values for the first dimension. For example, \( \llk \text{null, null, } \mu_1\rr_{\text{FrPhil}[\_1]} \) is a one dimensional array of FrPhil.

### 3.3 Run-time exceptions

Run-time exceptions are thrown when run-time integrity is violated. For example, dereferencing of null causes the exception NullPointerException, assignment to array components results in
the exception `ArrayStoreException` if the value is not compatible with the run-time type of the components of the array, etc. Therefore, in the operational semantics we have rules like:

\[
\begin{align*}
\epsilon[k] &= \text{val}, \sigma \vdash \text{throw new Nullable()}, \sigma \\
\text{if } \sigma(i) &= \{\text{val}_0, \ldots, \text{val}_{n-1}\}^{[i]} \ldots [n] \text{ and } \\
\text{val does not fit } T[1] \ldots [n] \text{ in } P, \sigma.
\end{align*}
\]

3.4 Descriptions, typing, compilation, and link-time checks

Compilation of a program, and more specifically type checking, requires type information about all classes and interfaces in this program and in any other programs used in this program. A Java compiler extracts the required information from the currently compiled program as well as from the *\textit{.class} files.

We formalize this type information as descriptions. Descriptions contain the names of classes/interaces, their superclasses, superinterfaces, the names of fields and their types, the names of methods and return types, argument types, and exception classes declared in throws clauses. Descriptions do not contain the method bodies and therefore correspond to C++ declarations.

Descriptions are designated as \( D, D', D_1, \ldots \). For example, \( D_{ph} \), the description of the philosophers example is in section 5.2, and the descriptions \( D_{st}, D_{cs}, D_{ab}, D_{sv}, D_{ba} \) are given in figure 32.

For typing we also need the types of this, the local variables and the parameters. These are stored in environments, which we denote as \( V, V', V_1, \ldots \).

Typing has the general format:

\[
\text{Descriptions} \times \text{Environments} \vdash \text{term : Type}
\]

Because we distinguish three languages, we define three judgements for typing, i.e., \( D, V \vdash t^a : T \), \( D, V \vdash t^b : T \), \( D, V, \sigma \vdash t^c : T \), where \( t^a \) is a Java\(^a\) term, \( t^b \) is a Java\(^b\) term, \( t^c \) is a Java\(^c\) term, \( T \) is a type, and \( \sigma \) is a state. Note that the state \( \sigma \) is required for the types of Java\(^c\) terms, because these may contain addresses.

Thus, compilation checks the types using the descriptions:

\[
\mathcal{C} : \text{Descriptions} \times \text{Environment} \times \text{Java}^a \to \text{Java}^b
\]

For example, \( \mathcal{C}(D_{ph}, e, S_{ph}) = B_{ph} \), and \( \mathcal{C}(D_{ph}D_{cs}, e, S_{cs}) = B_{cs} \), where \( D_{ph} \) is the description of the standard classes outlined in figure 2; \( S_{ph}, B_{ph} \) can be found in sections 4, 7; \( S_{cs}, B_{cs} \) can be found in figures 30, 31.

We can prove that compilation preserves types, and that Java\(^b\) types are preserved by the Java\(^a\) type system, that is:

\[
\begin{align*}
&D, V \vdash t^a : T \implies D, V \vdash t^b : T, \\
&D, V \vdash t^b : T \implies \forall \text{states } \sigma : D, V, \sigma \vdash t^c : T.
\end{align*}
\]

In order to describe separate compilation, we need to be able to extract descriptions out of compiled code. These will be augmented by the descriptions from the current program. Therefore we define the description extraction function:

\[
\mathcal{D} : (\text{Java}^a\text{-program} \to \text{Descriptions}) \cup (\text{Java}^b\text{-program} \to \text{Descriptions})
\]

With this we can define compilation \( \mathcal{C}^2 \) of a Java\(^a\)-program in the context of Java\(^b\)-programs:

\[
\begin{align*}
\mathcal{C}^2 : & \text{Java}^a \times \text{Java}^b \to \text{Java}^b \\
\mathcal{C}^2[S, B] &= \mathcal{C}(\mathcal{D}(B) \circ \mathcal{D}(S), e, S)
\end{align*}
\]

That is, we compile \( S \) using the descriptions from \( B \) updated by the descriptions from \( S \). The operation \( D_1 \circ D_2 \) represents augmenting the descriptions in \( D_1 \) by those in \( D_2 \); besides, the descriptions in \( D_2 \) override those in \( D_1 \) which introduce classes and interfaces of the same names. For example, \( D_{st}D_{cs} \circ D_{cv}D_{ba} = D_{st}D_{cv}D_{ba} \) and \( D_{bat}D_{cv} \circ D_{cs}D_{ba} = D_{bat}D_{cs}D_{ba} \).
The above approach differs from our previous \cite{7,8} where the descriptions and environments were given independently of the program. It also differs from \cite{37} where typing is defined with respect to a complete Java$_{Int}$ (corresponding to our Java\textsuperscript{b}) program. None of these approaches allows the description of separate compilation – although it must be noted that research did not aim to describe separate compilation.

Finally, in order to study issues around compilation, link-time checks and binary compatibility, we want to represent a high-level view of the bytecode and associated checks. We claim that Java\textsuperscript{b} and the associated typing are appropriate for these purposes.

Namely, although most current Java implementations use the bytecode \cite{21}, any code satisfying the requirements of the Java specification may be used. Java\textsuperscript{b} does contain all the information required for execution and is near to the programmer’s view on Java. The linking checks described in some detail in chapter 12.3 of \cite{14} consist of verification, preparation and resolution. Verification and preparation ensure that the code is structurally correct and prepare some tables. Resolution replaces symbolic references to fields and methods of other classes by more direct references \cite{21}. In Java\textsuperscript{b} we do not distinguish between symbolic references and actual fields or methods. Thus, we do not represent the replacement process, but we distinguish between successful resolution and resolution with errors through the Java\textsuperscript{b} type system. Namely, the judgement $\judge {\mathcal {B}} {\mathcal {D}}$ expresses that \textit{B} is well-formed, and that resolution was completed without errors, \textit{i.e.} all external references have been resolved.

Thus, we have a high-level view of linking and link-time checks. However, we have not yet modelled the dynamic aspect of linking, a task that we plan to tackle soon.

3.5 Normal and abnormal types

As we demonstrated through the $\P_{\text{act}}$ example in section 2.3, term execution may proceed normally, where any exception thrown is caught, or abnormally, where some exception is thrown but not caught. For example, $\text{peter.act()}$ might execute normally, or it may execute abnormally and throw an exception of class \textit{Worry}. Note, that the case where $\text{peter.act()}$ throws an \textit{Illness} exception and subsequently catches it belongs to normal execution.

A term may continue execution normally or abnormally. Therefore, it has a \textit{normal type}, which characterizes its normal execution, and an \textit{abnormal type}, which characterizes its abnormal execution \cite{10}. A term whose execution will continue abnormally only has the normal type $\bot$. Normal types are primitive types, interfaces, classes, \texttt{void}, or $\bot$; these are denoted by $\mathcal {T}$, $\mathcal {T}'$, $\mathcal {T}_i$, \textit{etc}. Abnormal types are sets of subclasses of the predefined class \texttt{Throwable}; those are denoted by $\mathcal {E}_1$, $\mathcal {E}_2$, $\mathcal {E}_3$, or $\{\mathcal {E}_1, \ldots, \mathcal {E}_n\}$, \textit{etc}. Notice, that normal types may contain subclasses of \texttt{Throwable}. Such cases represent the situation where an object of a subclass of \texttt{Throwable} is returned but not thrown. For example, \texttt{Illness}, \texttt{Person}, \texttt{void} are normal types, \{\\texttt{Illness}, \texttt{Worry}\}, \{\texttt{NullPointerException}, \texttt{Worry}\} are abnormal types.

We define six different type inference systems which describe the normal and abnormal types of Java\textsuperscript{a}, Java\textsuperscript{b} and Java\textsuperscript{c} terms. Thus, the corresponding six judgements are $\mathcal {D}, \mathcal {V} \vdash_{\mathcal {N}}^\mathcal {T} t : \mathcal {T}$, $\mathcal {D}, \mathcal {V} \vdash_{\mathcal {N}}^{\mathcal {E}_1} t : \mathcal {E}_1$, $\mathcal {D}, \mathcal {V} \vdash_{\mathcal {N}}^{\mathcal {E}_2} t : \mathcal {E}_2$, $\mathcal {D}, \mathcal {V} \vdash_{\mathcal {N}}^{\mathcal {E}_3} t : \mathcal {E}_3$, $\mathcal {D}, \mathcal {V}, \sigma \vdash_{\mathcal {N}}^\mathcal {T} t : \mathcal {T}$, $\mathcal {D}, \mathcal {V}, \sigma \vdash_{\mathcal {N}}^{\mathcal {E}_1} t : \mathcal {E}_1$, $\mathcal {D}, \mathcal {V}, \sigma \vdash_{\mathcal {N}}^{\mathcal {E}_2} t : \mathcal {E}_2$, $\mathcal {D}, \mathcal {V}, \sigma \vdash_{\mathcal {N}}^{\mathcal {E}_3} t : \mathcal {E}_3$. These judgements may be distinguished by the annotations on the symbol $\vdash$. Namely the subscripts $\mathcal {N}$ and $\mathcal {E}$ distinguish between the normal and abnormal types, and the superscripts $\mathcal {N}$, $\mathcal {E}_1$, $\mathcal {E}_2$, $\mathcal {E}_3$ distinguish between the three languages. For example, for $\mathcal {D}_{\text{act}}$ – the description of the worries example $\P_{\text{act}}$ – and the environment $\mathcal {V}_{\text{act}} = \texttt{Person peter}$; we have the following typings:

$$
\begin{align*}
\mathcal {D}_{\text{act}}, \mathcal {V}_{\text{act}} \vdash_{\mathcal {N}}^\mathcal {T} \text{peter.act()} : \texttt{void}, \\
\mathcal {D}_{\text{act}}, \mathcal {V}_{\text{act}} \vdash_{\mathcal {E}_1}^\mathcal {T} \{\texttt{Worry, Illness}\}, \\
\mathcal {D}_{\text{act}}, \mathcal {V}_{\text{act}} \vdash_{\mathcal {N}}^{\mathcal {E}_1} \text{peter.diagnostics()} : \texttt{Illness}, \\
\mathcal {D}_{\text{act}}, \mathcal {V}_{\text{act}} \vdash_{\mathcal {E}_1}^{\mathcal {E}_1} \text{peter.diagnostics()} : \{\texttt{Worry}\}, \\
\mathcal {D}_{\text{act}}, \mathcal {V}_{\text{act}} \vdash_{\mathcal {E}_1}^{\mathcal {E}_2} \text{throw new Worry} : \bot, \\
\mathcal {D}_{\text{act}}, \mathcal {V}_{\text{act}} \vdash_{\mathcal {E}_1}^{\mathcal {E}_3} \text{throw new Worry} : \{\texttt{Worry}\}.
\end{align*}
$$

To obtain the type of a term we combine the normal type $\mathcal {T}$ with the abnormal type $\mathcal {E}_1$ as $\mathcal {T} \parallel \mathcal {E}_1$. Thus,
### 3.6 Widening and subject reduction

Widening corresponds to subtyping, and it is required for assignments and for parameter passing. A normal type widens to another normal type if they are identical, or if the first is a subclass or subinterface of the latter. For example,

\[ D_p \vdash \text{Int} \leq_w \text{Int} \]

and this makes the assignment \( a : \text{Int} \rightarrow \text{Int} \), type-correct.

As we showed earlier, the type of a term consists of its normal and abnormal types. A type widens to another type if the normal type of the first is \( \bot \), or widens to the normal type of the latter, and if every class in the abnormal of the first either is an unchecked exception (i.e., subclass of Error or RuntimeException) or has a superclass in the abnormal type of the latter. For example,

\[ D_v \vdash \text{Illness} \} \leq_w \text{Illness} \} \{ \text{Worry}, \]

\[ D_v \vdash \bot \} \{ \text{Worry} \} \leq_w \text{Illness} \} \{ \text{Worry}, \]

\[ D_v \vdash \bot \} \{ \text{Worry} \} \leq_w \text{void} \} \{ \text{Worry, Illness} \}

\[ D_v \vdash \text{void} \} \{ \text{Worry, Illness} \} \leq_w \text{void} \} \{ \text{Throwable}. \]

The subject reduction theorem states that execution of a term produces a new term whose type widens to that of the original. Therefore, the theorem guarantees that normal execution returns a value compatible with the normal type of the original term, and abnormal execution terminates with an exception compatible with the abnormal type of the term. For example, the typing

\[ D_{v^*} \vdash \text{pete纠纷}().\text{treat}() : \text{Illness} \} \{ \text{Worry} \]

guarantees that execution of the term \( \text{pete纠纷}().\text{treat}() \) will either return an object of class \( \text{Illness} \) or throw an exception of class \( \text{Worry} \) (or a subclass), or it will throw an unchecked exception such as \( \text{NullPointerException} \) or \( \text{OutOfMemoryError} \).

### 4 The language Java

Java\(^*\) describes a subset of Java. It includes classes, interfaces, instance variables, instance methods, inheritance of instance methods and variables, hiding of instance variables, overloading and overriding of methods, widening, method calls, assignments, object creation, the null value, field access and the exception class \( \text{Null} \) and the exception classes \( \text{ArrayIndexOutOfBoundsException}, \text{NegativeArraySizeException} \) and \( \text{IllegalArgumentException} \). Although arrays are objects, we distinguish between them because of the extra operations available on arrays. 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, \( \text{super} \), strings, numeric promotions and widenings, concurrency, and packages.

There are slight differences between the syntax of Java\(^*\) and Java. We introduced these differences in order to simplify the description. Firstly, \texttt{ext} represents \texttt{extends}, \texttt{impl} represents \texttt{implements}. Separators like \{ \}, ( ), , and ; are omitted when these are obvious. The implicit Java direct superclass \texttt{Object} has to be specified explicitly in Java\(^*\). Also, the \texttt{impl} and \texttt{throws} clauses are not optional, i.e., they may be followed by empty lists. Finally, the Java syntax for
conditionals, i.e., if (Expr) Stmts else Stmts, is represented in Java* as if Expr then Stmts else Stmts.

We follow the convention that Java* separators, operator symbols and keywords appear as **keywords**, 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.

A Java* program, as described in figure 1, consists of a **sequence** of class definitions or interface definitions. A class definition introduces a new class as a subclass of another class, lists the interfaces implemented by that class and contains field or method definitions. A field definition consists of a field name and a type. A method definition consists of a method header and a method body. A method header includes the return type, a method identifier, identifiers and types of formal parameters, and possibly an empty list of class names in its **throws** clause. A method body consists of a statement sequence. We require that there is at most one **return** statement in each method body and that if there is one, it should return an expression and it should be the last statement. This simplifies our operational semantics without restricting the expressiveness, since it requires minor transformations to enable any Java method body to satisfy this property. An interface definition introduces a new interface as a subinterface of several other interfaces and as a sequence of method headers.

| Program  ::=  Def*                           
| Def     ::=  class ClassId ext ClassName impl InterfName*             
|          { ClassMember* }                     
|          | interface InterfId ext InterfName* { InterfMember* }             
| ClassMember ::=  Field | Method                   
| Field    ::=  VarType VarId;                       
| Method   ::=  MethHeader MethBody                
| Meth.Header ::=  ( void | VarType | MethId ((VarType ParId)*) throws ClassName*           
| Meth.Body ::=  [Stmts [return Expr] ]            
| 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]+([[]]) | this              
| Var      ::=  Name | Expr.VarName | Expr[Expr]                              
| Value    ::=  PrimValue | RefValue                                 
| RefValue ::=  null                                  
| PrimValue ::=  intValue | charValue | boolValue | ...                       
| VarType  ::=  SimpleType | ArrayType                                  
| SimpleType ::=  PrimType | ClassName | InterfName                                
| ArrayType ::=  SimpleType[] | ArrayType[]                                 
| PrimType  ::=  bool | char | int | ...                                

Figure 1: Java* programs

The Java statements we consider are conditional statements, assignments, method calls, return, try, and throw statements. This is sufficient because loop and case statements can be coded in terms of conditionals and recursion.

The Java expressions we consider are values, object creation, array creation, method call, field access, array access, and this. Values in Java are primitive values or references. Because references
appear only at run-time, in Java\textsuperscript{8} we can express only those values that can be specified by means of literals. Therefore Java\textsuperscript{8} values are primitive (e.g., literals such as true, false, 3, 'c', etc.) and the null reference (represented by the null literal). The expression new C() creates a new object of class C, whereas the expression new T[a1]...[a2][1]...[k], n ≥ 1, k ≥ 0 creates a n+k-dimensional array value.

\[
S_8 = \begin{align*}
class \ Object \ ext \ Object \ impl \ { & ... \} \\
class \ Throwable \ ext \ Object \ impl \ { & ... \} \\
class \ Error \ ext \ Throwable \ impl \ { & ... \} \\
class \ Exception \ ext \ Throwable \ impl \ { & ... \} \\
class \ RTE \ ext \ Exception \ impl \ { & ... \} \\
class \ NullPE \ ext \ RTE \ impl \ { & ... \} \\
class \ ArrStoreE \ ext \ RTE \ impl \ { & ... \} \\
class \ IndOutBndE \ ext \ RTE \ impl \ { & ... \} \\
class \ NegSzeE \ ext \ RTE \ impl \ { & ... \} \\
\end{align*}
\]

Figure 2: The Java\textsuperscript{8} basic program

The basic program \(S_8\), defined in figure 2, is the Java\textsuperscript{8} program that contains the basic classes and interfaces described in chapters 20-22 of [14]. In particular, for our current work \(S_8\) includes the classes Throwable, Error, Exception, and the runtime exception class RTE with the subclasses NullPE, ArrStoreE, IndOutBndE, and NegSzeE.

The Java philosophers program \(P_{ph}\) from section 2.1 corresponds to the following Java\textsuperscript{8} program

\[
S_{ph} = \begin{align*}
class \ Quest \ ext \ Object \ impl \ { & } \\
class \ Exist \ ext \ Quest \ impl \ { & } \\
class \ Phil \ ext \ Object \ impl \ { & } \\
& \text{Truth like;} \\
& \text{Book think(Quest y) throws \{"thinkBody1": \ldots\}} \\
& \text{FrPhil think(Exist y) throws \{"thinkBody2": \ldots\}} \\
\} \\
class \ FrPhil \ ext \ Phil \ impl \ { & } \\
& \text{Food like;} \\
& \text{Book think(Quest y) throws \{this.like=oyster;\ldots\}} \\
\} \\
class \ Test \ ext \ Object \ impl \ { & } \\
& \text{void test(Quest aQuest, Exist being, Phil aPhil, FrPhil pascal) throws \{} \\
& \text{\ldots} \\
& \}}
\]

5 Type notions, descriptions, and environments

5.1 Type notions

Figure 3 describes type notions of the Java language, which do not appear as such in Java programs. Argument types describe the types of arguments required in a method header. Argument types are tuples of variable types, which may be primitive types, interfaces, classes or arrays as defined in figure 1.

Normal types indicate the type of possible results of a term under normal execution. A normal type may be a variable type, nil, \texttt{void}, or \perp. The normal type \perp describes terms which will
continue execution abnormally only. Note that $\bot$ does not stand for the error. It stands for a well-typed term whose execution will be abnormal. Abnormal types describe the exceptions escaping from a method under abnormal execution; they consist of sets of subclasses of the Throwable class. The type $\bot \parallel \emptyset$ never occurs for well-typed expressions.

5.2 Descriptions

Descriptions are defined in figure 4. They consist of typing information about each interface, class and their instance variables and methods. Descriptions of a program consist of class and interface descriptions. A class description consists of the class identifier, names of superclasses and superinterfaces, types and identifiers of fields, and method headers. A method header includes the method identifier, return type, argument types and identifiers, and throws clause. An interface description consists of the interface identifier, names of its superinterfaces and its method headers. Remember, that Field and MethHeader have been defined in figure 1.

![Figure 4: Descriptions](image)

The mapping $D$ extracts the descriptions of Java$^a$ or Java$^b$ programs (defined in section 7):

$$D : (\text{Java}^a\text{-program} \rightarrow \text{Descriptions} | \cup \text{Java}^b\text{-program} \rightarrow \text{Descriptions})$$

$D$ can be naturally extended to any composition of Java$^a$ or/and Java$^b$ programs:

$$D : (\text{Java}^a\text{-program} | \text{Java}^b\text{-program})^* \rightarrow \text{Descriptions}$$

The description of an interface can be obtained from the interface definition without any modifications. The description of a class can be obtained from the class definition by ignoring the method bodies.

**Definition 1** For a Java$^a$ or Java$^b$ program $P$ the description mapping $D$ is defined as follows:

$$D(\text{interface } I \text{ ext } I_1, \ldots, I_j \{\text{meth}_1, \ldots, \text{meth}_k\} \ P) =$$

$$D(\text{class } C \text{ ext } C' \text{ impl } I_1, \ldots, I_j \{\text{cMem}_1, \ldots, \text{cMem}_k\} \ P) =$$

$$D(\text{cMem}) = \begin{cases} \text{true} & \text{if } \text{cMem} = \text{true} \\ \text{meth} & \text{if } \text{cMem} = \text{meth}\text{.mBody}, \text{ where meth is a method header} \\ \text{Undefined} & \text{otherwise} \end{cases}$$

$$D(\epsilon) = \epsilon$$
Notice that definition 1 applies to both Java\(^a\) and Java\(^b\) programs.

Descriptions are denoted by \(D, D_1, D'\), etc. Corresponding to the basic program \(S_b\) (defined in figure 2) the basic descriptions \(D_b = D(S_b)\) contain the descriptions of all classes and interfaces of \(S_b\). The descriptions of the philosophers program is the following:

\[
D_{ph} = D(S_{ph}) =
\begin{align*}
\text{class} & \text{ Quest ext Object } \\
\text{class} & \text{ Exist ext Quest } \\
\text{class} & \text{ Phil ext Object impl }
\begin{align*}
& \text{Truth like:} \\
& \text{Book think(Quest y) throws ;} \\
& \text{FrPhil think(Exist y) throws ;}
\end{align*}
\text{class} & \text{ FrPhil ext Phil impl }
\begin{align*}
& \text{Food like:} \\
& \text{Book think(Quest y) throws ;}
\end{align*}
\text{class} & \text{ Test ext Object impl }
\begin{align*}
& \text{void test(Quest aQuest, Exist being, Phil aPhil, FrPhil pascal) throws ;}
\end{align*}
\end{align*}
\]

The descriptions for the students examples, \(D_{at}, D_{cs}, D_{cfr}, D_{lab}, D_n\), can be found in figure 32.

Type checking takes place in the context of the descriptions of several classes and interfaces. For example, we would check the class FrPhil11 in the context of the descriptions \(D_{ph}\) and \(D_n\).

The following four judgements can be made about descriptions:

1. \(\vdash D \diamond u\), i.e. \(D\) is unambiguous. Descriptions are unambiguous if every interface, class, and field has a unique description, as can be established according to the rules in figure 5. (Methods are not unique in Java; these can be overloaded; the requirements on well-formed methods are given in the corresponding rules for class and interface in figures 9 and 13.) For unambiguous descriptions we define various look-up functions in sections 5.3, 5.4. These functions are used for establishing well-formedness of class and interface descriptions in figure 9.

2. \(D' \vdash D \diamond u\), i.e. \(D\) is acyclic in \(D'\). This means that the class and interface hierarchies of the entities introduced in \(D\) are acyclic in \(D'\). This is described in figure 6 in section 5.4. For \(D\) with \(D' \vdash D \diamond u\) the functions which look-up class and interface members are well-defined. These functions are used for establishing well-formedness of class and interface descriptions as described in section 5.6 and for typing, as described in sections 6, 9, 12.

3. \(D' \vdash D \diamond\), i.e. \(D\) is well-formed in the context of \(D'\). Well-formedness of descriptions means that (1) \(D\) is unambiguous; (2) \(D\) is acyclic in \(D'\); (3) the classes and interfaces in \(D\) have types at the declarations of superclasses and superinterfaces, fields, method parameters and return and throwable results, and the restrictions applying to overloading and overriding are satisfied; (4) these types have their declarations in \(D\) or \(D'\). This is described in figure 9 in section 5.6.

4. \(\vdash D \diamond\), i.e. \(D\) is complete. This means that \(D\) is well-formed by itself, i.e. it contains all type information necessary to establish its well-formedness. This is described in figure 9 in section 5.6.

Note that when we write the acyclic and well-formed judgements in a short form like \(D' \vdash D \diamond u\) and \(D' \vdash D \diamond\), the descriptions \(D'\) contains the description \(D\). For the philosophers example \(D_{ph} \vdash D_{ph} \diamond\), holds, but \(D_n \vdash D_{ph} \diamond\) does not hold. For the students example \(D_1 D_{at} D_{cfr} \vdash D_{cfr} \diamond\) holds but it is impossible to establish that \(D_{at} \vdash D_{cs} \diamond\) or \(D_{ph} D_{at} D_{cs} \vdash D_{cfr} \diamond\). Note also, that \(D' \vdash D \diamond\) does not imply \(D' \vdash D' \diamond\)!
5.3 Unambiguous descriptions

Figure 5 describes the requirements for descriptions to be unambiguous. They ensure that interfaces, classes, and fields in classes are unique.

\[
\forall i \in \text{interface } i \text{ ext } i_1, \ldots, i_s \{ \ldots \} D_2, \\
D = D_3 \text{ interface } i \text{ ext } i_1', \ldots, i_s' \{ \ldots \} D_4 \\
\Rightarrow D_1 = D_3, D_2 = D_4 \text{ and } D \neq D_b \text{ class } i \text{ ext } \{ \ldots \} D_6 \\
\forall c \in \text{class } c \text{ ext } c \implies i_1, \ldots, i_s \{ \ldots \} D_2, \\
D = D_3 \text{ class } c \text{ ext } c' \implies i_1', \ldots, i_s' \{ \ldots \} D_4 \\
\Rightarrow D_1 = D_3, D_2 = D_4 \\
\forall f \in \text{class } c \text{ ext } c \{ \text{def } f_1 \text{ T f; def } f_2 \} D_2, \\
D = D_3 \text{ class } c \text{ ext } c' \{ \text{def } f_3 \text{ T' f; def } f_4 \} D_4 \\
\Rightarrow \text{def } f_1 = \text{def } f_3, \text{def } f_2 = \text{def } f_4
\]

Figure 5: Unambiguous descriptions

For \(\vdash D \circ \sigma\) the functions \(I\text{Dec}(D, I), CD\text{Dec}(D, C), F\text{Dec}(D, C, f), M\text{Dec}(D, X, m)\) look up the description \(D\) to find interface \(I\), class \(C\), field \(f\) in a given class and method \(m\) in a given class or interface \(X\), respectively. The functions \(\text{SuperC}(D, C), \text{Interfs}(D, X)\) look up the direct superclass of the class \(C\) and all direct superinterfaces of class or interface \(X\).

In order to extract the method header components, namely, a method identifier, argument types, return type, and classes of \(\text{throws}\) clause from a method description, we introduce the functions \(\text{MethId}(\_), ArgT(\_), RetT(\_), ExcT(\_))\) first. These are used to indicate a method name in the method look up function (in definition 3) and to describe the type of a method call (figures 11, 12, 17, 18, 28, 29) as well as other restrictions (cf. figure 9), which are imposed on methods (see section 5.6), as given in chapters 8.2 and 9 in [14].

**Definition 2** For a method header \(\text{methH}\) such that

\[
\text{methH} = T \#(T_1 \text{ p}_1, \ldots, T_n \text{ p}_n) \text{ throws } E_1, \ldots, E_q
\]

we define the following functions:

\[
\text{MethId}(\text{methH}) = \# \\
\text{ArgT}(\text{methH}) = T_1 \times \ldots \times T_n \\
\text{RetT}(\text{methH}) = T \\
\text{ExcT}(\text{methH}) = \{E_1, \ldots, E_q\}
\]

Note that the range of function \(\text{ArgT}(\text{methH})\) is \(\text{ArgType}\), and the range of function \(\text{ExcT}(\text{methH})\) is \(\text{AbnormType}\) as defined in figure 3.

**Definition 3** For descriptions \(D\) and identifiers \(I, C, X, m, f\) we define the following look-up functions:
\[ IDecl(D, I) = \begin{cases} 
\text{interface } I \ ext \ I_1,...,I_j \ \{\text{Decl}\} & \text{iff } D = D' \ \text{interface } I \ ext \ I_1,...,I_j \ \{\text{Decl}\} \ D'' \\
\text{Undef} & \text{otherwise}
\end{cases} \]

\[ CDecl(D, C) = \begin{cases} 
\text{class } C \ ext \ C' \ \text{impl} \ I_1,...,I_j \ \{\text{Decl}\} & \text{iff } D = D' \ \text{class } C \ ext \ C' \ \text{impl} \ I_1,...,I_j \ \{\text{Decl}\} \ D'' \\
\text{Undef} & \text{otherwise}
\end{cases} \]

\[ SuperC(D, C) = \begin{cases} 
C' & \text{iff } CDecl(D, C) = \text{class } C \ ext \ C' \ \ldots\ \{\ldots\} \\
\text{Undef} & \text{otherwise}
\end{cases} \]

\[ Interfs(D, X) = \begin{cases} 
\{I_1,...,I_j\} & \text{iff } IDecl(D, X) = \text{interface } X \ ext \ I_1,...,I_j \ \{\ldots\} \\
\emptyset & \text{or } CDecl(D, X) = \text{class } X \ \ldots \ \text{impl} \ I_1,...,I_j \ \{\ldots\} \\
\text{Undef} & \text{otherwise}
\end{cases} \]

\[ MDecl(D, X, m) = \begin{cases} 
\{\text{mthH} | IDecl(D, X) = \text{interface } X \ \ldots \ \{\text{mthH}\} \ \ldots \ \text{and} \ \text{MethId}(\text{mthH}) = m\} & \text{iff } IDecl(D, X) \neq \text{Undef} \\
\emptyset & \text{otherwise}
\end{cases} \]

\[ FDecl(D, C, f) = \begin{cases} 
T & \text{iff } CDecl(D, C) = \text{class } C \ \ldots \ \{T \ f; \ \ldots\} \\
\text{Undef} & \text{otherwise}
\end{cases} \]

If \( D \vdash C \circ u \), then the look-up functions are well-defined.

5.4 Acyclic descriptions, look-up, widening

The subclass, \( \sqsubseteq \), implements, \( \lhd \), and subinterface, \( \leq \), relationships deduced from descriptions \( D \) are defined by the inference rules in figure 6.

The superinterface of an interface is indicated in its declaration. The assertion \( D \vdash I \leq I' \) means that \( I \) is a subinterface of \( I' \). The subinterface relationship is transitive. Each interface is its own subinterface, thus the assertion \( D \vdash I \leq I \) indicates that \( I \) is declared in \( D \) as an interface.

The direct superclass of a class is indicated in its description. The assertion \( D \vdash C \sqsubseteq C' \) indicates that \( C \) is a subclass of \( C' \). Each class in \( D \) is its own subclass, therefore, the assertion \( D \vdash C \sqsubseteq C \) indicates that \( C \) is declared as a class in \( D \). The subclass relationship is transitive.

The assertion \( D \vdash C : \circ \ _{\ldots} \ I \) indicates that the class \( C \) is declared in \( D \) as providing an implementation for the interface \( I \).

A description \( D \) is acyclic in \( D' \), i.e., \( D'D \vdash D \circ_a \), if the subclass and subinterface relationships introduced in \( D \) are acyclic in \( D' \). Note that \( D'D \vdash D \ sqsubseteq_a \ D' \) does not imply that \( D \vdash D' \sqsubseteq_a \), nor that \( D \vdash D \ sqsubseteq_a \). However, \( D'D \vdash D \ sqsubseteq_a \) and \( D'D' \vdash \ D' \ sqsubseteq_a \) imply \( D'D' \vdash D \ sqsubseteq_a \) and \( D'D' \vdash D' \ sqsubseteq_a \). Notice also that \( D \vdash X \ sqsubseteq_a \) implies that \( X \) is declared as a class or interface in \( D \).

**Definition 4** \( D \vdash C \sqsubseteq C' \sqsubseteq C'' \) iff \( D \vdash C \sqsubseteq C' \) and \( D \vdash C' \sqsubseteq C'' \).

The following field and method look-up functions are defined in the class/interface hierarchy. The functions search in a given class or interface \( X \), and if the searched field or method is not found, they search in the superclass or superinterface. So, these functions are defined only if the class/interface hierarchy for \( X \) is acyclic, i.e., \( D \vdash X \ sqsubseteq_a \). Thus, for acyclic descriptions the following functions extract the members of a class or interface:

- \( Fields(D, C, f) \) indicates all field definitions for \( f \) (namely, the class containing the field declaration, the type and the name of the field), which were defined in \( C \) or in a superclass of \( C \) and possibly hidden by \( C \) or another superclass;
\[ D = D' \text{ interface } I \text{ ext } \ldots I', \ldots \{ \ldots \} D'' \]

\[ D \vdash I \leq I' \]

\[ D \vdash I \leq I' \]

\[ D = D' \text{ class } C \text{ ext } C' \text{ impl } \ldots I', \ldots \{ \ldots \} D'' \]

\[ D \vdash \text{ nil} \subseteq C, \ D \vdash C \subseteq C, \ D \vdash C \subseteq C', \ D \vdash C : \text{ imp } I \]

\[ \text{IDec}(D, I') \neq \text{Undef}, \ D \vdash I' \odot_a \]

\[ D \vdash I' \leq I \]

\[ D = D' \text{ interface } I \text{ ext } \ldots I', \ldots \{ \ldots \} D'' \]

\[ \text{IDec}(D, I) \neq \text{Undef}, \ D \vdash I \odot_a \]

\[ \text{CDec}(D, C) \neq \text{Undef}, \ D \vdash C \odot_a \]

\[ \text{IDec}(D, I) \neq \text{Undef} \implies D' D \vdash I \odot_a \]

\[ \text{CDec}(D, C) \neq \text{Undef} \implies D' D \vdash C \odot_a \]

\[ D' D \vdash \text{D } \odot_a \]

Figure 6: Subclasses, subinterfaces and acyclic descriptions

- \textit{Meths}(D, X, m)\) indicates all method declarations for a method \(m\) in \(X\) or inherited from one of its superclasses or superinterfaces and not hidden by any of those.

Hidden fields are treated differently from overridden methods. Namely, hidden fields are part of the set \( \text{Fields} \), whereas overridden methods are not part of the set \( \text{Meths} \). The reason for the difference is that hidden fields 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 hidden in FrPhil), whereas overridden methods are never called by objects of the subclasses (e.g. for FrPhil objects the only think method with a Quest argument is that from FrPhil, whereas that defined in Phil is of no interest to FrPhil objects).

**Definition 5** For descriptions \( D \), identifiers \( X, C, m, f \) with \( D \vdash X \odot_a \) and \( D \vdash C \odot_a \) we define the members look-up as follows:

\[
\text{Fields}(D, C, f) = \begin{cases} \{ (C \ f) \} \cup \text{Fields}(D, \text{Super } C(D, C), f) & \text{iff } \text{FDec}(D, C, f) = T \ f \\ \text{Fields}(D, \text{Super } C(D, C), f) & \text{iff } \text{FDec}(D, C, f) = \text{Undef} \end{cases}
\]

\[
\text{Meths}(D, X, m) = \begin{cases} \text{MDec}(D, X, m) \cup \{ \text{meth}H' \mid \text{meth}H' \in \text{Meths}(D, \text{Super } C(D, X), m) \text{ and } \forall \text{meth}H \in \text{MDec}(D, X, m) \ n T(\text{meth}H') \neq \text{ArgT}(\text{meth}H) \} & \text{iff } D \vdash X \subseteq X \\ \text{MDec}(D, X, m) \cup \{ \text{meth}H' \mid \exists \text{I}' \in \text{Interfs}(D, X) \ n \text{meth}H' \in \text{Meths}(D, \text{I}', m) \text{ and } \forall \text{meth}H \in \text{MDec}(D, X, m) \ n T(\text{meth}H') \neq \text{ArgT}(\text{meth}H) \} & \text{iff } D \vdash X \subseteq X \end{cases}
\]

Note that even though \( \text{Super } C(D, \text{Object}) = \text{Object} \), the above look-up functions are well-defined.

For the philosophers example \( P_{\text{ph}} \) the above functions are:

21
\[
\begin{array}{ll}
\text{Fields}(D, \text{Dp}, \text{Phil}, \text{fun}) & = \emptyset \\
\text{Fields}(D, \text{Dp}, \text{Phil}, \text{like}) & = \{ \text{Phil Truth like} \} \\
\text{Fields}(D, \text{Dp}, \text{FrPhil}, \text{like}) & = \{ \text{Phil Truth like}, \text{FrPhil Food like} \} \\
\text{Melts}(D, \text{Dp}, \text{Phil}, \text{play}) & = \emptyset \\
\text{Melts}(D, \text{Dp}, \text{Phil}, \text{think}) & = \{ \text{Book think(Quest y) throws} , \\
& \quad \text{FrPhil think(Exist y) throws} \} \\
\text{Melts}(D, \text{Dp}, \text{FrPhil}, \text{think}) & = \{ \text{Book think(Quest y) throws} , \\
& \quad \text{FrPhil think(Exist y) throws} \}
\end{array}
\]

\[
\begin{array}{c}
\frac{D \vdash \text{int} \diamond V_{\text{arType}}}{D \vdash I \leq I} \\
\frac{D \vdash \text{char} \diamond V_{\text{arType}}}{D \vdash C \subseteq C} \\
\frac{D \vdash \text{bool} \diamond V_{\text{arType}}}{D \vdash C \diamond V_{\text{arType}}} \\
\frac{D \vdash T_i \diamond V_{\text{arType}}}{D \vdash T_i \in \{1\ldots n\}, n \geq 0} \\
\frac{D \vdash T \times \ldots \times T_n \diamond \text{ArgType}}{D \vdash T \diamond V_{\text{arType}}} \\
\frac{D \vdash T \diamond \text{RetType} \text{ or } T = \text{void}}{D \vdash T \diamond \text{RetType} \text{ or } T = \text{nul} \text{ or } T = \bot} \\
\frac{D \vdash E_i \subseteq \text{Throw} \quad j \in \{1\ldots q\}, q \geq 0}{D \vdash \{E_1, \ldots, E_q\} \diamond \text{AbsType}} \\
\frac{D \vdash T \diamond \text{AbsType}}{D \vdash \text{ET} \diamond \text{AbsType}} \\
\frac{D \vdash \text{ET} \diamond \text{AbsType}}{D \vdash T \parallel \text{ET} \diamond \text{Type}}
\end{array}
\]

Figure 7: Well-formed types

Figure 7 defines well-formedness for types. The assertions \( D \vdash T \diamond V_{\text{arType}} \), \( D \vdash \text{AT} \diamond \text{ArgType} \), \( D \vdash T \diamond \text{RetType} \), \( D \vdash T \diamond \text{AbsType} \), \( D \vdash \text{ET} \diamond \text{AbsType} \), \( D \vdash T \parallel \text{ET} \diamond \text{Type} \) mean correspondingly that \( T \) is a variable type, \( \text{AT} \) is an argument type, \( T \) is a return type, \( T \) is a normal type, \( \text{ET} \) is an abnormal type, \( T \parallel \text{ET} \) is a type.

In order to reason about abnormal types as sets of exception classes, we introduce the more specific relation \( \subseteq_e \) and the \textit{difference} operation \( \setminus_e \) in definition 6. The relation \( \subseteq_e \) is the set-theoretic subset taking the subclass relationship into account. The relation \( \setminus_e \) is the set-theoretic difference taking the subclass relationship into account.

**Definition 6** For descriptions \( D \) with \( D \vdash \text{ET} \diamond \text{AbsType} \) and \( D \vdash \text{ET}' \diamond \text{AbsType} \) we define \( \subseteq_e \) and \( \setminus_e \) as follows:

- \( D \vdash \text{ET} \subseteq_e \text{ET}' \iff \forall E \in \text{ET} \exists E' \in \text{ET}' : D \vdash E \sqsubseteq \text{Error} \text{ or } D \vdash E \sqsubseteq \text{RTE} \text{ or } D \vdash E \sqsubseteq E' \)
- \( \text{ET} \setminus_e \text{ET}' = \{ E' \mid E' \in \text{ET}' \text{ and } \forall E \in \text{ET} \not\vdash E' \sqsubseteq E \} \)

The more specific relation is reflexive, transitive and antisymmetric. It is used in order to describe whether the throws clause of a method header covers all possible exceptions thrown in its body and in any overriding method (see figure 9 and lemma 1). Also it is used in the definition of widening relationship over types (see figure 8). The difference operation is used to determine which exceptions may escape out of try-catch-finally statements (see figures 12, 18, 29). For the worries example (section 2.3) and the corresponding descriptions \( D_e \)
\[ D_{\text{w}} \vdash \{ \text{Worry, Illness} \} \subseteq_{w} \{ \text{Exception} \} \]
\[ \{ \text{Worry} \} \setminus_{\text{w}} \{ \text{Exception} \} = \emptyset \]
\[ \{ \text{Exception} \} \setminus_{\text{w}} \{ \text{Worry, Illness} \} = \{ \text{Exception} \} \]

Widening, \( \leq_{w} \), is the reflexive projection of the subclass and subinterface relations to the domains of normal types and of types. It is defined over normal types, argument types, and types.

\[
\begin{align*}
T \neq \bot & \quad D \vdash T \odot_{\text{NorType}} \quad D \vdash T \leq_{w} \text{Object} \\
D \vdash T \leq_{w} T & \quad D \vdash \text{nil} \leq_{w} T \\
D \vdash T \leq T & \quad D \vdash T \subseteq T' \\
D \vdash T \leq_{w} T' & \quad D \vdash T \subseteq_{w} T' \\
D \vdash T : \text{imp} T'' & \quad D \vdash T \odot_{\text{VarType}} \\
D \vdash T'' \leq_{w} T''' & \quad D \vdash T'' \leq_{w} T'''
\end{align*}
\]

\[
D \vdash T_{1} \leq_{w} T'_{1} \quad i \in \{1 \ldots n\}, \quad n \geq 1 \\
D \vdash T_{1} \times \cdots \times T_{n} \leq_{w} T'_{1} \times \cdots \times T'_{n}
\]

\[
D \vdash T \leq_{w} T' \quad \text{or} \quad T = \bot \quad \text{and} \quad D \vdash T' \odot_{\text{NorType}} \\
D \vdash E \subseteq_{w} E' \\
D \vdash E \sqcup E \leq_{w} E' \sqcup E'
\]

Figure 8: Widening over normal types, argument types, types

Widening is expected between the left-hand and right-hand side of assignments (e.g., figures 11, 17, 28) and also between the actual and formal parameters of method calls (e.g., definition 9 and figure 11). For these cases we need widening over variable types. This is defined in figure 8 by the assertion \( D \vdash T \leq_{w} T' \). The type \text{nil} can be widened to any array, class or interface.

The widening relation between variable types is naturally projected to argument types (see the ninth rule of figure 8) and thus allows an elegant description of applicable and more special methods in definition 9.

The widening relation for types is defined in the last rule in figure 8. This relation is used, in particular, to describe the type of a method in well-formed programs (e.g., figure 13), taking into account both its return type and the exceptions mentioned in the \texttt{throws} clause. Also, we use widening for types in the proof of the subject reduction theorem (section 11).

Note that widening is not reflexive for \( \bot \), i.e., \( D \vdash \bot \not\leq_{w} \bot \). This has purely technical reasons and allows more concise proof of subject reduction.

5.5 Properties of acyclic descriptions

It is straightforward to state and prove the following properties of acyclic descriptions.

Two variable types that are in the subclass relationship are classes: \( \subseteq_{w} \) is reflexive, transitive and antisymmetric; the subclass hierarchy forms a tree. Also, two variable types that are in the

\footnote{Chapter 5.1.2 in [14] defines also widening over primitive types, e.g., \texttt{int} widens to \texttt{char}, but we do not describe such cases here.}
subinterface relationship are interfaces; ≤ is transitive, reflexive and antisymmetric. Unlike ⊆, ≤
does not form a tree. Two abnormal types that are in the more specific relationship are sets of
exception classes; ≤ is reflexive, transitive and antisymmetric.

Widening is reflexive, transitive and antisymmetric.

If an interface widens to a variable type, then the variable type is a superinterface of
the interface or it is the class Object. If an interface widens to a variable type (which is different from
Object), then either the interface is identical to the variable type, or one of interface’s immediate
superinterfaces is a subinterface of the variable type.

If a variable type widens to a class, then the variable 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 argument type widens to another argument type, then the first argument type consists of
variable types which widen to the variable types of the second argument type.

5.6 Well-formed and complete descriptions, and their properties

Figure 9 defines well-formedness of descriptions. A description D is well-formed in the context
of another description D', if D is unambiguous and acyclic in D' and if all the class and interface
descriptions in D are well-formed.

A class or interface description is well-formed, iff

- its superclass is class, and its superinterfaces are interfaces;
- fields, method parameters and results have well-formed types;
- instance methods with the same identifier have different argument types;
- an overriding method has the same return type as the overridden method;
- an overriding method throws only such checked exception classes which are subclasses of
  those specified in the throws clause of any overridden method;
- “unless a class ... is abstract, the declarations of methods defined in each direct superinterface
  are implemented either by a declaration in this class, or by an existing method declaration
  inherited from a direct superclass”.

Descriptions are complete, i.e. ⊢ D ⊢, if they are well-formed in their own context, i.e. if they
contain all information required for checking.

The following lemma states that if a variable type T is widened to another variable type T'
and T' has a method m, then there exists in T a unique method m with the same argument types,
same return type and more specific abnormal type than the method from T'.

**Lemma 1** For descriptions D with ⊢ D ⊢, if D ⊢ T ≤w T', meth' ∈ Meths(D,T',m) then
exists methH ∈ Meths(D,T,m) so that ArgT(methH) = ArgT(meth'H), RetT(methH) = RetT(meth'H),
D ⊢ ExcT(methH) ≤e ExcT(meth'H).

5.7 Environments

As usual, for typing we need the notion of an environment, which maps variables to their types.
Environments are used for typing method bodies and contain the types of the formal parameters
and the type of the receiver of a method, i.e. this.

An environment, defined in figure 10, consists of declarations for variables and/or for this.
Environments are usually designated by V, V', V₁, etc. An environment is well-formed, i.e. D ⊢ V ⊢, if the declarations are unambiguous, and their types are well-formed in D. In well-formed
environments look-up of variables, V[v], or of this, V[this], gives their types.

So, type checking or compilation of the method test from the philosopher example (section 2.1)
may take place in the environment:

Vₚ₉ = Test this; Quest aQuest; Exist being; Phil aPhil; FrPhil pascal;
\[
\begin{align*}
D & \vdash D \otimes \\
\vdash & D \otimes u \\
D' & \vdash D \otimes u \\
\forall I & \; I'D \not\equiv \text{Decl}(D, I) \implies D' \vdash \text{Decl}(D, I) \otimes \\
\forall C & \; C'D \not\equiv \text{Decl}(D, C) \implies D' \vdash \text{Decl}(D, C) \otimes
\end{align*}
\]

\[
\begin{align*}
I' & \in \text{Interfs}(D, I) \implies D \vdash I' \leq I' \\
\text{methH} & \in \text{Decl}(D, I, m) \\
& \implies D \vdash \text{Ret}(\text{methH}) \otimes \text{RetType}, \\
& D \vdash \text{Arg}(\text{methH}) \otimes \text{ArgType}, \\
& D \vdash \text{Exc}(\text{methH}) \otimes \text{ExcType} \\
\text{methH}_1 & \in \text{Decl}(D, I, m) \text{ and } \text{methH}_2 \in \text{Decl}(D, I', m) \text{ and } \text{methH}_1 \neq \text{methH}_2 \\
& \implies \text{Arg}(\text{methH}_1) \neq \text{Arg}(\text{methH}_2) \\
\forall m, \forall I' & \in \text{Interfs}(D, I), \\
\text{methH} & \in \text{Meths}(D, I, m), \text{methH}' \in \text{Meths}(D, I', m), \text{Arg}(\text{methH}) = \text{Arg}(\text{methH}') \\
& \implies \text{Ret}(\text{methH}) = \text{Ret}(\text{methH}'), D \vdash \text{Exc}(\text{methH}) \subseteq e \text{ Exc}(\text{methH}')
\end{align*}
\]

\[
\begin{align*}
D & \vdash \text{Decl}(D, I) \otimes \\
C' & = \text{SuperC}(D, C) \implies D \vdash C' \subseteq C' \\
I & \in \text{Interfs}(D, C) \implies D \vdash I \leq I \\
T & = \text{FDecl}(C, D, T) \implies D \vdash T \otimes \text{VarType} \\
\text{methH} & \in \text{Decl}(D, C, m) \text{ with } \text{methH} = T \equiv (T_1, p_1, \ldots, T_n, p_n) \text{ throws } E_1, \ldots, E_q \\
& \implies D \vdash T \otimes \text{RetType}, \\
& D \vdash T \times \ldots \times T \otimes \text{ArgType}, \\
& D \vdash (E_1, \ldots, E_q) \otimes \text{ExcType} \\
\text{methH}_1 & \in \text{Decl}(D, C, m) \text{ and } \text{methH}_2 \in \text{Decl}(D, C, m) \text{ and } \text{methH}_1 \neq \text{methH}_2 \\
& \implies \text{Arg}(\text{methH}_1) \neq \text{Arg}(\text{methH}_2) \\
\forall m & \text{methH} \in \text{Decl}(D, C, m), \text{methH}' \in \text{Meths}(D, C', m), \text{Arg}(\text{methH}) = \text{Arg}(\text{methH}') \\
& \implies \text{Ret}(\text{methH}) = \text{Ret}(\text{methH}'), D \vdash \text{Exc}(\text{methH}) \subseteq e \text{ Exc}(\text{methH}') \\
\forall m, \forall I & \in \text{Interfs}(D, C), \text{methH}_1 \in \text{Decl}(D, I, m) \\
& \implies \text{methH}_2 \in \text{Meths}(D, C, m), \text{Arg}(\text{methH}_1) = \text{Arg}(\text{methH}_2), \\
& \text{Ret}(\text{methH}_1) = \text{Ret}(\text{methH}_2), D \vdash \text{Exc}(\text{methH}_2) \subseteq e \text{ Exc}(\text{methH}_1)
\end{align*}
\]

Figure 9: Well-formed and complete descriptions
6 The Java\textsuperscript{a} type system

Because types are normal - abnormal type pairs, there are two kinds of typing judgements for Java\textsuperscript{a} terms: \( D, V \vdash^a T : T \) and \( D, V \vdash^a T : ET \) which establish the normal type \( T \) and the abnormal type \( ET \).

6.1 Normal types for Java\textsuperscript{a} terms

Figure 11 describes normal types for Java\textsuperscript{a} terms.

According to the rules for literals and identifiers, boolean literals have boolean type, character literals have character type, integer literals have integer type, identifiers have the types as declared in the environment. The null literal has the nil type.

In order to describe a normal type for statement sequence and conditional, we need to define two operations \( \cap \) and \( \cup \).

**Definition 7** For descriptions \( D \) with \( D \vdash^a T \), \( T \vdash^a T' \). We define operations \( \cap \), \( \cup : \text{NormType} \times \text{NormType} \rightarrow \text{NormType} \):

\[
\begin{align*}
T \cap T' &= \begin{cases} 
\top & \text{iff } T = \top \\
\bot & \text{otherwise}
\end{cases} \\
T \cup T' &= \begin{cases} 
\bot & \text{iff } T = \bot \text{ and } T' = \bot \\
\text{void} & \text{iff } (T = \text{void} \text{ or } T' = \text{void}) \text{ and } T, T' \in \{\text{void}, \bot\}
\end{cases}
\end{align*}
\]

Thus, a statement sequence has the same type as its last statement, provided that \( \bot \) is not the type of the first statements. Otherwise the sequence has type \( \bot \). A conditional consists of two statement sequences whose type may be either \textbf{void} or \( \bot \). If either branch has type \textbf{void}, then the conditional has the type \textbf{void}, otherwise it has the type \( \bot \).

The Java requirement that no statement should be unreachable (chapter 14.19 of [14]) could be modelled by forbidding a statement sequence \( \text{stat} \ast \text{stat} \) with \( \text{stat} \) of normal type \( \bot \), as this would have made \( \text{stat} \ast \text{stat} \) unreachable. Alternatively, as we discuss later we could require method bodies not to have normal type \( \bot \). However, since this requirement is not crucial for type soundness and since it does not hold for Java\textsuperscript{a} statement sequence – namely, exception throwing and propagation turn part of the code into unreachable statement – we do not model the reachability requirement.

According to the rule for assignment, an expression of normal type \( T' \) can be assigned to a variable of normal type \( T \) if \( T' \) can be widened to \( T \).

A return statement has the same type as the expression it returns.
\begin{itemize}
  \item \texttt{i is integer, c is character. D \vdash V \diamond, V(x) = T x}
  \item D, V \vdash \texttt{true} : \texttt{bool}
  \item D, V \vdash \texttt{false} : \texttt{bool}
  \item D, V \vdash i \in \texttt{int}, D, V \vdash c \in \texttt{char}, D, V \vdash x : T
  \item D, V \vdash e : \texttt{bool}
  \item D, V \vdash \texttt{stats} : T
  \item D, V \vdash \texttt{stats'} : T'
  \item D, V \vdash \texttt{stmt} : T''
  \item D, V \vdash \texttt{if} \ e \ \texttt{then} \ \texttt{stats} \ \texttt{else} \ \texttt{stats'} : T \ \texttt{∪} \ T'
  \item D, V \vdash \texttt{v} : T
  \item D, V \vdash e : T'
  \item D \vdash \ T' \ \leq \ T
  \item D, V \vdash e : \texttt{void}
  \item D, V \vdash \texttt{e} : T
  \item D, V \vdash \texttt{return} \ e : T
  \item D \vdash \ C \ \sqsubseteq \ C
  \item D, V \vdash \texttt{new} \ C() : C
  \item D \vdash \ T \ \diamond \ V_{arType}
  \item D, V \vdash \texttt{e_1} : \texttt{int}, i \in \{1...n\}, n \geq 1, k \geq 0
  \item D, V \vdash \texttt{new} \ T[e_1][...][e_m][...][k] : T[1]...[k+n]
  \item D \vdash \ T \ \diamond \ a
  \item D, V \vdash \texttt{e} : T
  \item D, V \vdash \texttt{e}' : \texttt{int}
  \item FirstVis(D, T, f) = (C T' f)
  \item D, V \vdash \texttt{e}_1 : T
  \item D, V \vdash e : E
  \item D \vdash \texttt{E} \ \sqsubseteq \ \texttt{Throwable}
  \item D, V \vdash \texttt{throw} \ e : \bot
  \item n \geq 0
  \item D \vdash \texttt{E}_i \ \sqsubseteq \ \texttt{Throwable} \ i \in \{1...n\}
  \item D \vdash \ V \ \diamond
  \item V(v_1) = \texttt{Undef} \ i \in \{1...n\}
  \item D, V \vdash \texttt{E}_i \ v_1 : \texttt{stmt}_1 : T_i \ i \in \{1...n\}
  \item D, V \vdash \texttt{stmts}_0 : T_0
  \item D, V \vdash \texttt{stmts}_{i+1} : T_{i+1}
  \item D, V \vdash \texttt{try} \ \texttt{stmts} \ \texttt{catch} \ \texttt{E}_i \ v_1 \ \texttt{stmts}_i ... \ \texttt{catch} \ E_m \ v_m \ \texttt{stmts}_m : \texttt{void}
  \item D, V \vdash \texttt{try} \ \texttt{stmts}_0 \ \texttt{catch} \ \texttt{E}_i \ v_1 \ \texttt{stmts}_i ... \ \texttt{catch} \ E_m \ v_m \ \texttt{stmts}_m
  \item \texttt{finally} \ \texttt{stmts}_{m+1} : T_{m+1}
\end{itemize}

Figure 11: Normal types for Java\textsuperscript{a} terms
The next group of the rules describes the type of object and array creation. For a class \( C \), the expression `new C()` has type \( C \). For a simple type \( T \), the expression `new T[x1]...[x_n]` is an \( n+k \)-dimensional array of elements of type \( T \).

Below, the rules describe the type of field and array access. For an array access \( e[a] \) the expression \( e \) should have an array type \( T [] \), and \( a \) should be of integer type. For a field access \( e.f \) the expression \( e \) should have a class type \( T \), one of whose superclasses \( C \) should contain a field declaration for \( f \) as of type \( T' \). Moreover, it should be the first visible field declaration, i.e. \( f \) is declared in a superclass \( C \) and there is no other class \( C' \) between the original class \( T \) and the superclass \( C \) which contains another field declaration for \( f \).

**Definition 8** For descriptions \( D \), an identifier \( m \), classes \( C, C' \), \( T \) with \( D \vdash T \land m \), variable type \( T' \):

\[
\text{FirstVis}(D, T, f) = (C T' f) \iff (C T' f) \in \text{Fields}(D, T, f) \quad \text{and} \quad \forall C' \neq C \quad D \vdash T \supseteq C' \supseteq C \quad \implies \quad \text{FDec}(D, C', f) = \text{Undef}
\]

If the field declaration \( T' f \) is first visible from class \( C \) in the class hierarchy, i.e. \( \text{FirstVis}(D, T, f) = (C T' f) \), then the field access expression has the type \( T' \).

The method call type rule of figure 11 describes method call, as in ch. 15.11 [14]. A method is applicable if the actual parameter types can be widened to the corresponding formal parameter types. A method signature is more special, \( \ll \), than another method signature, 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 method signatures are the minima of the more special partial order.

**Definition 9** For descriptions \( D \), identifiers \( m \), \( T \) with \( D \vdash T \land m \), argument type \( AT \):

- \( \text{AppMeths}(D, m, AT) = \{ (T', \text{methH}) \mid \text{methH} \in \text{MDec}(D, T, m) \quad \text{and} \quad D \vdash T \subseteq_w T' \quad \text{and} \quad D \vdash \text{ArgT(methH)} \leq_w AT \} \)

- \( D \vdash (T, \text{methH}) \ll (T', \text{methH'}) \quad \text{iff} \quad D \vdash T \subseteq_w T' \quad \text{and} \quad D \vdash \text{ArgT(methH)} \leq_w \text{ArgT(methH')} \)

- \( \text{MostSpec}(D, m, T, AT) = \{ \text{methH} \mid (T, \text{methH}) \in \text{AppMeths}(D, m, T, AT) \quad \text{and} \quad (T', \text{methH'}) \in \text{AppMeths}(D, m, T, AT) \quad \text{and} \quad D \vdash (T', \text{methH'}) \ll (T, \text{methH}) \implies (T', \text{methH'}) = (T, \text{methH}) \} \)

The most special applicable methods are contained in the set \( \text{MostSpec} \). A method call is type-correct if this set contains exactly one method header. The argument type of this method header is stored as the method descriptor, c.f. ch. 15.11 in [14] and the rule for a method call in figure 15. The return type of the method header is the return type of the method call.

The last two groups of rules in figure 11 describe the normal type for statements dealing with exception throwing and catching. A throw statement has the type \( \bot \), if the expression following the throw indicates an exception. The try-catch statement has the type \( \text{void} \), provided that the constituent statement lists are well-typed, and that the names of exception classes and new variables appear after each catch.\(^3\) Under the same conditions, the try-catch-finally has the same type as the statements comprising its finally block. The additional Java requirements, that no class \( E_i \) should appear more than once, and that no class should be followed by a subclass expressed in [19] are omitted here, since they do not affect the subject reduction property. Thus, for the example \( P \), the normal types of the statements `throw diagnose().treat();` is \( \bot \), and `try { if (...) throw diagnose().treat(); else age=age+1;}catch(Illness i){i.cure();}` is \text{void}.

### 6.2 Abnormal types for Java\(^a\) terms

Figure 12 describes abnormal types for Java\(^a\) terms.

\(^3\) A more precise type than \( \text{void} \) could be given estimating cases where it could be guaranteed that none of handlers would catch exceptions definitely thrown in the try block. However the estimate will always remain conservative. That is why we give the simple type rule with \( \text{void} \) normal type.
| i is integer, c is character, D ⊨ V ◦, V(x) \neq \text{Undef} |
| D, V \|_0 \quad \text{true} : \emptyset | D, V \|_0 \quad \text{false} : \emptyset |
| D, V \|_0 \quad i : \emptyset | D, V \|_0 \quad c : \emptyset | D, V \|_0 \quad x : \emptyset |
| D, V \|_0 \quad \text{e} : \text{ET} |
| D, V \|_0 \quad \text{stmts} : \text{ET}' \quad D, V \|_0 \quad \text{stmts}' : \text{ET}'' \quad D, V \|_0 \quad \text{stmt} : \text{ET}'' |
| D, V \|_0 \quad \text{if e then stmts else stmts'} : \text{ET} \cup \text{ET}' |
| D, V \|_0 \quad \text{v} : \text{ET} |
| D, V \|_0 \quad \text{e} : \text{ET}' |
| D, V \|_0 \quad \text{return e : \text{ET}} |
| D, V \|_0 \quad \text{new C()} : \emptyset |
| D, V \|_0 \quad \text{new T[e_1][\ldots][e_n][\ldots][e_k]} : (\text{ET}'_{i-1}) |
| D, V \|_0 \quad \text{e} : \text{ET} |
| D, V \|_0 \quad \text{e'} : \text{ET}' |
| D, V \|_0 \quad \text{stmt} : \text{ET} \cup \text{ET}' |
| D, V \|_0 \quad \text{e.f} : \text{ET} |
| D ⊨ T_0 \circ_0 |
| D, V \|_0 \quad e_1 : \text{ET}_i \quad i \in \{0\ldots n\} \quad n \geq 0 |
| D, V \|_0 \quad e_1 : \text{T}_i \quad i \in \{0\ldots n\} \quad n \geq 0 |
| \text{MostSpec(D, } m, T_0, T_1 \times \ldots \times T_n = \{\text{methH}} |
| D, V \|_0 \quad \text{stmt}_0, \ldots, \text{stmt}_n : (\text{ET}'_{i-1}) \cup \{\text{E} \circ_0 T(\text{methH})} |
| e \neq \text{null} |
| D, V \|_0 \quad \text{e} : \text{E} |
| D, V \|_0 \quad \text{e} : \text{ET} |
| D, V \|_0 \quad \text{throw e : \{E\} \cup \text{ET}} |
| D, V \|_0 \quad \text{throw null : \{NullPE}} |

Figure 12: Abnormal types for Java\textsuperscript{8} terms
According to the rules for literals and identifiers, the literals of primitive types, the null literal, and identifiers have the empty abnormal type. According to the remaining rules, the abnormal type of a term is the union of the abnormal types of its subterms. For example, the abnormal type of statement sequence is the union of the abnormal types of the constituent statements. We do not consider constructors, that is why the abnormal type for the expression \( \text{new} \ C() \) does not include the abnormal type of a constructor and therefore is empty. For the same reason, the abnormal type of the expression for array creation, \( \text{new} \ T[e_1 \ldots e_n] \), consists only of the abnormal types of the expressions \( e_1, \ldots, e_n \).

The abnormal type of a method call is the union of the abnormal types of the receiver and the arguments and of the exception classes in the throws clause of the method header.

The last group of rules describe abnormal types for statements dealing with exception throwing and catching. A throw statement contains an expression; evaluated it will create and throw an exception object. The abnormal type of a throw statement followed by an expression different from null is the union of the normal type and the abnormal type of the expression; namely, evaluation of this expression might throw other exceptions. The abnormal type of a throw statement followed by null is the set of the predefined run-time exception \( \text{NullPE} \).

Therefore, for the exception example \( E \), the abnormal type of the statement throw \( \text{diagnose}() \cdot \text{treat}() \) is \( \{\text{Worry}, \text{Illness}\} \). The abnormal type of the try-catch-finally statement consists of the abnormal type of the try, catch and finally statements excluding the exception classes \( E_1, \ldots, E_n \) which are caught. Thus, for the worries example (section 2.3) and its description \( D_E \) the abnormal type of the statement try \{ if (...) throw \( \text{diagnose}() \cdot \text{treat}() \); else \( \text{age} = \text{age} + 1; \) \} catch \( \{ \text{I} \} \{ \text{cure}(); \} \)

\( \{\text{Worry}, \text{Illness}\} \cup \{\text{Worry}\} \) and the abnormal type of the statement try \{ act(); \} catch \( \{ \text{Throwable} \} \{ \text{age} = \text{age} + 2; \} \) is \( \{\text{Worry}, \text{Illness}\} \cup \{\text{Throwable}\} = \emptyset \).

### 6.3 Properties of the Java\textsuperscript{a} type system

Notice that we have no subsumption rule in our system. The subsumption rule says that any expression of type \( T \) also has type \( T' \) if \( T \) is a subtype of \( T' \). In the case of Java, where subtypes are expressed by the \( \leq_w \) relation, it would have had the form:

\[
\begin{align*}
D, V \vdash e & : T \\
D \vdash T & \leq_w T' \\
D, V \vdash e & : T'
\end{align*}
\]

The type system introduced in this paper does not obey the subsumption rule. For instance, the type of aPhil like is Truth, but the type of pascal like is Food, although \( D_{\text{ph}} \), \( V_{\text{ph}} \), \( p_{\text{ph}} \) pascal : FrPhil and \( D \vdash \text{FrPhil} \leq_w \text{Phil} \).

In fact, introduction of the subsumption rule would make this type system non-deterministic. Note however, that \( \mathfrak{P} \) 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.

**Lemma 2** \( D, V \models T \vdash \bot \parallel ET \implies ET \neq \emptyset \)

**Definition 10** \( D, V \models T \parallel ET \iff D, V \models T \quad \text{and} \quad D, V \models t : ET \)

**Definition 11** For a Java\textsuperscript{a} term \( t \), program \( P \), environment \( V \) we define:

\[
\begin{align*}
S, V \models t & : T \\
S, V \models t : T \\
S, V \models t : ET \\
S \vdash V & : \emptyset
\end{align*}
\]

where \( D = D(S) \).

---

\textsuperscript{4}Here also, we give a conservative estimate; more precise ones are possible especially with higher order systems.
6.4 Well-formed Java\(^2\) programs

In order to describe well-formedness of programs and also the operational semantics (in section 11.2) we need properties deduced from programs, and look-up functions for method bodies.

First we introduce auxiliary functions to extract the method components, namely, a method identifier and an argument type (similar to those introduced for method descriptions in definition 2). These definitions are trivially applied to Java\(^2\) programs as well. Note that Java\(^2\) programs are defined in section 7.

Definition 12 For a method \(\text{meth}\) such that

\[
\text{meth} = T = (T_1, p_1, \ldots, T_n, p_n) \text{ throws } E_1, \ldots, E_q \{ \text{mBody} \}
\]

we define the following functions:

\[
\text{MethId(meth)} = m
\]

\[
\text{ArgT(meth)} = T_1 \times \ldots \times T_n
\]

All properties defined for descriptions are defined for the corresponding Java\(^2\) or Java\(^3\) programs.

Definition 13 For a Java\(^2\) or Java\(^3\) program \(P\) we define the following judgements:

- \(P \vdash C \subseteq C'\) iff \(D \vdash C \subseteq C'\),
- \(P \vdash I \leq I'\) iff \(D \vdash I \leq I'\),
- \(P \vdash ET \subseteq \epsilon ET'\) iff \(D \vdash ET \subseteq \epsilon ET'\),
- \(P \vdash T \leq_w T'\) iff \(D \vdash T \leq_w T'\),
- \(\text{Fields}(P, C, f) = \text{Fields}(D, C, f)\),

where \(D = T(P)\).

Definition 14 For a program \(P\), identifiers \(C, m\) we define the following look-up functions:

\[
\text{IDef}(P, I) = \begin{cases} \text{interface I ext I}_1, \ldots, I_j \{ \text{iDecl} \} & \text{iff } P = P' \text{ interface I ext I}_1, \ldots, I_j \{ \text{iDecl} \} P'' \\ \text{Undef} & \text{otherwise} \end{cases}
\]

\[
\text{CDef}(P, C) = \begin{cases} \text{class C ext C'} \text{ impl } I_1, \ldots, I_j \{ \text{cBody} \} & \text{iff } P = P' \text{ class C ext C'} \text{ impl } I_1, \ldots, I_j \{ \text{cBody} \} P'' \\ \text{Undef} & \text{otherwise} \end{cases}
\]

\[
\text{SuperC}(P, C) = \begin{cases} \text{C'} & \text{iff } \text{CDef}(P, C) = \text{class C ext C'} \ldots \{ \ldots \} \\ \text{Undef} & \text{otherwise} \end{cases}
\]

\[
\text{MDef}(P, C, m) = \begin{cases} \{ \text{meth} \mid \text{CDef}(P, C) = \text{class C} \ldots \{ \ldots \text{meth} \ldots \} \text{ and MethId(meth) = } m \} & \text{iff } \text{CDef}(P, C) \neq \text{Undef} \\ \emptyset & \text{otherwise} \end{cases}
\]

\[
\text{Meths}(P, C, m) = \text{MDef}(P, C, m) \cup \{ \text{meth}' \mid \text{meth}' \in \text{Meths}(P, \text{SuperC}(P, C), m) \text{ and } \forall \text{meth} \in \text{MDef}(P, C, m) \text{ ArgT(meth')} \neq \text{ArgT(meth)} \}
\]

\[
\text{FirstFit}(P, m, C, k) = \{ \text{meth} \mid \text{meth} \in \text{Meths}(P, C, m) \text{ and } \text{ArgT(meth)} = k \}
\]
\[
\begin{align*}
& D \vdash D(S) \circledast \\
& \forall I \text{ IDef}(S, I) \neq \text{Undef} \implies D \vdash \text{IDef}(S, I) \circledast \\
& \forall C \text{ CDef}(S, C) \neq \text{Undef} \implies D \vdash \text{CDef}(S, C) \circledast
\end{align*}
\]

Figure 13: Well-formed Java\textsuperscript{5} programs

Note that FirstFit(P, m, C, AT) returns the whole method including the method body, whereas FirstFit(D, m, C, AT), defined later in definition 15, returns a method header only.

As defined in figure 13, a Java\textsuperscript{5} program S is well-formed i.e. $D \vdash S \circledast$, iff $D \vdash D(S) \circledast$, i.e. its descriptions are well-formed, and $D \vdash \text{CDef}(S, C) \circledast$, i.e. all its classes are well-formed, which means that all the method bodies are well-formed, i.e. they return values or throw exceptions as declared in their headers.\textsuperscript{5} Notice that well-formedness of a program can only be established in a description that contains the description of the program and possibly others. For the philosophers example, we can establish $D_b D_{pb} \vdash S_{pb} \circledast$, but we can establish neither $D_{pb} \vdash S_{pb} \circledast$, nor $D_b \vdash S_{pb} \circledast$.

7 Java\textsuperscript{b}, enriching Java\textsuperscript{5}

Java\textsuperscript{b} is an enriched version of Java\textsuperscript{5} which provides compile-time type information necessary at run-time. The syntax of Java\textsuperscript{b} programs is defined in figure 14. The process of enriching Java\textsuperscript{5} terms is by the mapping C given in section 8.

There are two cases where Java\textsuperscript{b} is different from Java\textsuperscript{5}, namely, method call and instance variable access. Method calls are enriched by the argument types of the most special applicable method available at compile-time. Thus, the Java\textsuperscript{5} syntax $\text{Expr.MethName(Expr)}$ is replaced in Java\textsuperscript{b} by $\text{Expr.[ArgType][MethName(Expr)]}$. Instance variable accesses are enriched by a descriptor – the class containing the field declaration. $\text{Expr.VarName}$ is replaced in Java\textsuperscript{b} by $\text{Expr.[ClassName][VarName]}$.

The basic program $B_b$ corresponds to $S_b$ given in Java\textsuperscript{5} in section 4. It contains the same classes and interfaces as $S_b$ with enriched field accesses and method calls.

Examples of enriched method call and instance variable access can be found in the following $B_{pb}$ program – therein lies the difference between $S_{pb}$ and $B_{pb}$.

\begin{verbatim}
B_{pb} = ...

    class Phil ext Object impl { ...
    }

    class FrPhil ext Phil impl {
    
      Food like;

      Book think(Quest) throws { this.[FrPhil]like = oyster;... }

\end{verbatim}

\textsuperscript{5}We could also have enforced the requirement of [14], p. 205, that all the statement should be reachable, by adding to the premises of the second rule of figure 13 that $T' \neq \bot$. 

32
Program ::= Def*  
Def ::= class ClassId ext ClassName impl InterfName*  
  {ClassMember*}  
  | interface InterfId ext InterfName* {InterfMember*}  
ClassMember ::= Field | Method  
Method ::= MethHeader  
MethHeader ::= ( void | VarType | MethId ((VarType ParId)*) throws ClassName*  
MethBody ::= {Stmts [return Expr]}  
Stmt ::= if Expr then Stmts else Stmts | Var = Expr  
  | Expr.[ArgType]MethName(Expr*) | throw Expr  
  | try Stmts (catch ClassName Id Stmts)* finally Stmts  
  | try Stmts (catch ClassName Id Stmts)+  
Expr ::= Value | Var | Expr.[ArgType]MethName(Expr*)  
  | new ClassName() | new SimpleType([Expr]+([])*) | this  
Var ::= Name | Expr.[ClassName]VarName | Expr[Expr]  
Value ::= PrimValue | RefValue  
RefValue ::= null  
PrimValue ::= intValue | charValue | boolValue | ...  
VarType ::= SimpleType | ArrayType  
SimpleType ::= PrimType | ClassName | InterfName  
ArrayType ::= SimpleType[] | ArrayType[]  
PrimType ::= bool | char | int | ...  

Figure 14: Java\textsuperscript{b} programs
8 Compilation of Java\textsuperscript{s} programs

The compilation of Java\textsuperscript{s} programs is described by \( \mathcal{C} \) which maps the triplet, consisting of a description, an environment, and a Java\textsuperscript{s} term, to the enriched Java\textsuperscript{b} term:

\[
\mathcal{C} : \text{Descriptions} \times \text{Environment} \times \text{Java}^s \rightarrow \text{Java}^b
\]

The description contains the information from several classes and interfaces. For example, compilation of \( S_{\phi} \) is described as \( B_{\phi} = \mathcal{C}(D_{\phi}, \epsilon, S_{\phi}) \).

The environment contains the variables and their types, \textit{i.e.} the parameters and the receiver \texttt{(this)} with its type. Local variables should also be reflected in the environment, but we have not yet modelled these. For example, compilation of the body of the method \texttt{think} of class \texttt{FrPhil} is \( \mathcal{C}(D_{\phi}, \texttt{FrPhil this}, \texttt{Quest y}, \texttt{this}.like=\texttt{system}...) \).

In actual implementations compilation takes place interleaved with type checking. However, for reasons of clarity we describe compilation separately from type checking.

Compilation of Java\textsuperscript{s} expressions and statements is described in figure 15.

According to the first rule, compilation of literals leaves them unmodified. Compilation of all other terms consists of the compilation of their subterms except for field access (sixth line) and method call (seventh line).

Compilation of a field access \( \texttt{e.f} \) extends the expression by the descriptor \( C \), the name of the class from which the field definition is inherited. Compilation of method call extends the expression by the argument type found by the type checker. Note that for field access and method call we require the type of the receiver to be acyclic in order to guarantee that the look-up functions are well-defined.

Compilation of program entities is defined in figure 16.

For the compilation of a method body we require the body to be well-typed, and that it should have a type \( T' || ET' \) compatible to the result type of the method header \( T || ET \). Also, we require that compilation of a program proceeds only if all classes and interfaces are well-formed. These two restrictions ensure that compilation is defined only if the program is well-formed, \textit{i.e.} if it satisfies all compile-time checks.

Notice that even though methods defined in interfaces are never executed, we preserve them unmodified, because descriptions of interfaces are used for checking other parts of the code, \textit{i.e.} interfaces from a Java\textsuperscript{b} program \( B \) will be used in order to establish \( D(B), V \vdash \text{t : T} \).

9 The Java\textsuperscript{b} type system

After giving types to Java\textsuperscript{s} terms, we also give types to Java\textsuperscript{b} terms. However, the rationale for typing the two languages is different: Java\textsuperscript{s} typing corresponds to typing performed by a Java compiler, and it determines whether a term is well-formed. Java\textsuperscript{b} typing, on the other hand, does not correspond to compile-time type checking; it represents a high-level view of the type-checking performed by the verifier at the time of linking as we argue in [11, 9].

34
\[ D \vdash v \diamond \]

\[ C\{D, v, z\} = z \quad \text{if } z \text{ is integer, character, identifier, null, true, or false} \]

\[
C\{D, v, \text{stmts ; stmt}\} = C\{D, v, \text{stmts}\} ; C\{D, v, \text{stmt}\}
\]

\[
C\{D, v, \text{if } e \text{ then stmts else stmts}'\} = \\
\quad \text{if } C\{D, v, e\} \text{ then } C\{D, v, \text{stmts}\} \text{ else } C\{D, v, \text{stmts}'\}
\]

\[ C\{D, v, v=e\} = C\{D, v\} = C\{D, v, e\} \]

\[ C\{D, v, \text{return } e\} = \text{return } C\{D, v, e\} \]

\[ D \vdash C \in C \]

\[ C\{D, v, \text{new } C()\} = \text{new } C() \]

\[ D \vdash T \diamond \text{VarType} \]

\[ C\{D, v, \text{new } T[e_1]...[e_k][\|_1,...,\|_x]\} = \\
\quad \text{new } T[C\{D, v, e_1\}...[C\{D, v, e_k\}][\|_1,...,\|_x]\}
\]

\[ D \vdash T \diamond a \]

\[ D, v \models e : T \]

\[ \text{FirstVis}(D, T, f) = (C, T', f) \]

\[ C\{D, v, e.f\} = C\{D, v, e\} ; C[f] \]

\[ D \vdash T_0 \diamond a \]

\[ D, v \models e_i : T_i \quad i \in \{0...n\}, n \geq 0 \]

\[ \text{MostSpec}(D, m, T_0, T_1 \times ... \times T_n) = \{\text{methH}\} \]

\[ C\{D, v, e_0.m(e_1...e_k)\} = C\{D, v, e_0\}.[\text{ArgT(methH)} \mapsto \text{m}(C\{D, v, e_1\}...C\{D, v, e_k\})] \]

\[ C\{D, v, \text{throw } e\} = \text{throw } C\{D, v, e\} \]

\[ C\{D, v, \text{try stmts catch } E_1 \text{ v1 stmts1 ... catch } E_n \text{ vn stmtsn} \} = \]

\[ \quad \text{try } C\{D, v, \text{stmts}\} \text{ catch } E_1 \text{ v1 } C\{D, v, \text{stmts1}\} \ldots \]

\[ \quad \text{catch } E_n \text{ vn } C\{D, v, \text{stmtsn}\} \]

\[ C\{D, v, \text{try stmts catch } E_1 \text{ v1 stmts1 ... catch } E_n \text{ vn stmtsn finally stmtsnext}\} = \]

\[ \quad \text{try } C\{D, v, \text{stmts}\} \text{ catch } E_1 \text{ v1 } C\{D, v, \text{stmts1}\} \ldots \]

\[ \quad \text{catch } E_n \text{ vn } C\{D, v, \text{stmtsn}\} \text{ finally } C\{D, v, \text{stmtsnext}\} \]

---

**Figure 15:** Compilation of Java$^6$ terms
\[
\mathcal{C}[D, V, T; f] = T \land f
\]

\[D \vdash V; T_1, p_1; \ldots; T_n, p_n; \triangleleft \exists T', \exists T'' \mid D, V; T_1, p_1; \ldots; T_n, p_n; \mathbb{I} \mid \mathbb{Body} : T' \mid ET' \land \exists D \vdash T'' \mid ET' \leq \omega \mid E_1, \ldots, E_q \mid \mathcal{C}[D, V, T; p_1, \ldots; T_n, p_n; \mathbb{Body}] = \]

\[
T \equiv (T_1, \ldots, T_n, p_n) \mid \mathbb{Body} \mid E_1, \ldots, E_q \mid \mathcal{C}[D, V, T; p_1, \ldots; T_n, p_n; \mathbb{Body}] = \]

\[\mathcal{C}[D, e, \text{class } C \text{ ext } C' \mid \text{impl } I_1, \ldots, I_j \mid \{cMem_1 \ldots cMem_k\}] = \]

\[\text{class } C \text{ ext } C' \mid \text{impl } I_1, \ldots, I_j \mid \{C \{D, C \text{ this, cMem}_1 \ldots \}C \{D, C \text{ this, cMem}_2 \} \}
\]

\[\mathcal{C}[D, e, \text{interface } I \mid \text{ext } I_1, \ldots, I_j \mid \{\text{meth}H_1 \ldots \text{meth}H_k\}] = \]

\[\text{interface } I \mid \text{ext } I_1, \ldots, I_j \mid \{\text{meth}H_1 \ldots \text{meth}H_k\}
\]

\[D \vdash \mathbb{I} \mid \text{def} \_1, \ldots, \text{def}_n \triangleleft \]

\[\mathcal{C}[D, e, \text{def} \_1, \ldots, \text{def}_n] = \mathcal{C}[D, e, \text{def} \_1] \ldots \mathcal{C}[D, e, \text{def}_n]
\]

Figure 16: Compilation of Java^4 programs

A Java^b term that has emerged by enriching a well-typed Java^a term will be well-typed too, and will have the same type as the latter, c.f. lemma 5. The Java^b type rules are the same as the Java^a type rules, except where the expressions have different syntax.

The type rules for the compiled and enriched Java^b term t have the forms \( D, V \vdash_t^a t : T \) and \( D, V \vdash_t^b t : ET \) which describe the normal type T and the abnormal type ET of the term correspondingly.

Figure 17 describes the normal types for Java^b terms. Figure 17 and figure 11 have identical structure. There are only two rules which are different, namely, field access and method call.

The difference between the normal type of a field access in Java^a and Java^b is that in Java^b the normal type depends on the descriptor (i.e. C) instead of the normal type of the variable on the left of the field access (i.e. T).

Concerning Java^b method calls, we search for appropriate methods using the stored argument type \( (T_1 \times \ldots \times T_n) \) instead of the types of the actual expressions \( (T'_1 \ldots T'_n) \). For this search we first examine the class of the receiver expression for a method header with appropriate argument types, and then its superclasses.

**Definition 15** For a description D, identifiers m, T with \( D \vdash T \otimes_a \), argument type AT we define:

\[\text{FirstFit}(D, m, T; AT) = \{ \text{meth}H \mid \text{meth}H \in \text{Meths}(D, T, m) \text{ and } \text{ArgT}(\text{meth}H) = AT \}\]

Figure 18 describes the abnormal types for Java^b terms. The rules are identical to those for Java^a given in figure 12, with the only differences in the syntax of field access and method call.

**9.1 Properties of the Java^b type system**

The following lemma states that no more than one method header \( \text{meth}H \) with argument types AT can be found for a variable type T. The declaration of the method header will come from a superclass or superinterface of T. Also, any further subclasses of T will either inherit that method, or override it by another method with a header \( \text{meth}H \) which has the same arguments but may differ in \( \text{throws} \) part.
| i is integer, c is character, D ⊨ V \odot, V(x) = T x |
|---|---|
| D, V \mu e : bool, D, V \\null nil |
| D, V \mu i : int, D, V \mu c : char, D, V \\mu x : T |
| D, V \mu e : bool |
| D, V \mu \text{stats}: T, D, V \mu \text{stats’}: T’ \quad D, V \mu \text{stat}: T’’ |
| D, V \mu \text{stats}: \text{stat}: T \land T’ |
| D, V \mu \text{if \ e then \ stats \ else \ stats’}: T \lor T’ |
| D, V \mu v : T |
| D, V \mu e : T’ |
| D ⊨ T’’ \leq w, T |
| D, V \mu v = e : \text{void} |
| D, V \mu e : T |
| D, V \mu \text{return \ e}: T |
| n \geq 1, k \geq 0 |
| D \vdash T \odot VarType |
| D, V \mu e_i : \text{int} \ i \in \{1...n\} |
| D, V \mu \text{new \ C()}: C |
| D, V \mu \text{new \ T[e_1]...[e_k][i_1]...[i_k]: T[i_1]...[i_k]} |
| D \vdash T \odot a |
| D, V \mu e : T |
| D \vdash T \leq w, C |
| FirstVis(D,T,f) = (C T’ f) |
| D, V \mu e, [C]f : T |
| D \vdash T_0 \odot a |
| D, V \mu e_1 : T_1 \ i \in \{0...n\}, n \geq 0 |
| D \vdash T_1 \leq w, T_i \ i \in \{1...n\} |
| FirstVis(D,m,T_0,T_1 \times \ldots \times T_3) = \{\text{method}\} |
| D, V \mu e_0, [T_1 \times \ldots \times T_3][e_1...e_3] : \text{RetT(method)} |
| D, V \mu e : E |
| D \vdash E \odot \text{Throwable} |
| D, V \mu \text{throw \ e}: \perp |
| n \geq 0 |
| D \vdash E_i \odot \text{Throwable} \ i \in \{1...n\} |
| D \vdash V \odot |
| V(v_1) = \text{Undef} \ i \in \{1...n\} |
| D, V E_i \ v_i : B \ i \in \{1...n\} |
| D, V \mu \text{stmt}_1 : T_0 |
| D, V \mu \text{stmt}_1 : T_0 |
| D, V \mu \text{try \ stmt}_0 \ \text{catch} \ E_i \ v_i \ \text{stmt}_1 \ \ldots \ \text{catch} \ E_n \ v_n \ \text{stmt}_n : \text{void} |
| D, V \mu \text{try \ stmt}_0 \ \text{catch} \ E_i \ v_i \ \text{stmt}_1 \ \ldots \ \text{catch} \ E_n \ v_n \ \text{stmt}_n |
| \text{finally} \ \text{stmt}_n+1 : T_{n+1} |

Figure 17: Normal types for Java\textsuperscript{t} terms
\[ i \text{ is integer, } c \text{ is character, } D \vdash V \triangleleft, \ V(x) \neq \text{Undef} \]
\[
D, V \models_0 \text{true}: \emptyset; \quad D, V \models_0 \text{false}: \emptyset
\]
\[
D, V \models_0 \ l: \emptyset; \quad D, V \models_0 \ c: \emptyset; \quad D, V \models_0 \ x: \emptyset
\]
\[
D, V \models_0 \ e: \text{ET}
\]
\[
D, V \models_0 \ \text{stats}': \text{ET}'; \quad D, V \models_0 \ \text{stats}': \text{ET}''; \quad D, V \models_0 \ \text{stat} : \text{ET}'''
\]
\[
D, V \models_0 \ \text{if } e \text{ then } \text{stats } else \text{stats}': \text{ET} \cup \text{ET}' \cup \text{ET}''
\]
\[
D, V \models_0 \ v: \text{ET}
\]
\[
D, V \models_0 \ e: \text{ET}'
\]
\[
D, V \models_0 \ \text{return } e : \text{ET}
\]
\[
D, V \models_0 \ \text{new } C() : \emptyset
\]
\[
D, V \models_0 \ \text{new } T[e_1]...[e_n][l_1]...[l_k] : (\text{ET}_)_{l-1}
\]
\[
D, V \models_0 \ e: \text{ET}
\]
\[
D, V \models_0 \ e': \text{ET}'
\]
\[
D, V \models_0 \ e'[e'] : \text{ET} \cup \text{ET}'
\]
\[
D, V \models_0 \ e_1: \text{ET}_1, i \in \{1...n\}
\]
\[
D, V \models_0 \ \text{FirstFit}(D, m, T_0, T_1 \times ... \times T_n) = \{\text{methH}\}
\]
\[
D, V \models_0 \ e_1: \text{ET}_1, i \in \{0...n\}
\]
\[
D, V \models_0 \ e_1[T_1 \times ... \times T_n](e_1,...,e_n) : (\text{ET}_)_{l-1} \cup \text{ExcF}[\text{methH}]
\]
\[
e \neq \text{null}
\]
\[
D, V \models_0 \ e: \text{E}
\]
\[
D, V \models_0 \ e: \text{ET}
\]
\[
D, V \models_0 \ \text{throw } e : \{\text{ET}\}
\]
\[
D, V \models_0 \ \text{throw null} : \{\text{NullPE}\}
\]
\[
n \geq 0
\]
\[
D \vdash E_i \subseteq \text{Throwable}, \ i \in \{1...n\}
\]
\[
D \vdash V \triangleleft
\]
\[
V(v_i) = \text{Undef} , i \in \{1...n\}
\]
\[
D, V \vdash E_i v_i \models_0 \ \text{stats}_i : \text{ET}_i , i \in \{1...n\}
\]
\[
D, V \models_0 \ \text{stats}_0 : \text{ET}_0
\]
\[
D, V \models_0 \ \text{stats}_{n+1} : \text{ET}_{n+1}
\]
\[
D, V \models_0 \ \text{try } \text{stats}_0 \ \text{catch } E_1 v_1 \ \text{stats}_1... \text{catch } E_n v_n \ \text{stats}_n : \text{ET}_0 \setminus \text{ET}_1 \cup (\text{ET}_)_{l-1}
\]
\[
D, V \models_0 \ \text{try } \text{stats}_0 \ \text{catch } E_1 v_1 \ \text{stats}_1... \text{catch } E_n v_n \ \text{stats}_n \ \text{finally } \ \text{stats}_{n+1} : \text{ET}_0 \setminus \text{ET}_1 \cup (\text{ET}_)_{l+1}
\]

Figure 18: Abnormal types for Javab terms
Lemma 3 For descriptions \( D \) with \( \vdash D \diamond \), normal types \( T, T', T'' \), argument type \( AT \), and method header \( \text{methH} \):

- \( \text{card}(\text{FirstFit}(D, m, T, AT)) \leq 1 \)
- \( \text{FirstFit}(D, m, T, AT) = \{\text{methH}\} \implies \exists T' : D \vdash T \leq w T' \) and \( \text{methH} \in MDecl(D, T', m) \)
- \( \text{FirstFit}(D, m, T, AT) = \{\text{methH}\}, \text{and} \text{methH} \in MDecl(D, T', m) \) and \( D \vdash T'' \leq w T' \)
  \( \implies \exists T'', \text{methH}'' : \text{FirstFit}(D, m, T', AT) = \{\text{methH}''\}, \text{and} D \vdash T'' \leq w T' \) and
  \( \text{methH}'' \in MDecl(D, T'', m) \)

Definition 16 \( D, V \vdash t : T \mid ET \iff D, V \vdash t : T \) and \( D, V \vdash t : ET \)

Lemma 4 \( D, V \vdash t : t \mid ET \implies ET \neq \emptyset \)

Not surprisingly, when a well-typed Java\(^a\) term \( t \) of type \( T \) is enriched into a Java\(^b\) term, it retains its type. Also, compilation of \( t \) is well-defined if and only if \( t \) is well-typed.

Lemma 5 For descriptions \( D \), environment \( V \) with \( D \vdash V \diamond \), type \( T \), Java\(^a\) term \( t \), Java\(^b\) class or interface definition \( \text{def} \):

- \( D, V \vdash t : T \implies D, V \vdash t \in C \{D, V, t\} : T \)
- \( C \{D, \varepsilon, \text{def}\} \neq \text{UnDef} \iff D \vdash \text{def} \diamond \)

Definition 17 For a Java\(^b\) term \( t \), program \( P \), environment \( V \), we define

\[
\begin{align*}
B, V \vdash t & : T \iff D, V \vdash t : T, \\
B, V \vdash t & : T \iff D, V \vdash t : T, \\
B, V \vdash t & : ET \iff D, V \vdash t : ET, \\
B \vdash ET & \subseteq_e ET' \iff D \vdash ET \subseteq_e ET', \\
B \vdash q & \diamond \iff D \vdash q \diamond
\end{align*}
\]

where \( D = D(P) \).

9.2 Well-formed and complete Java\(^b\) programs

Figure 19 outlines well-formedness and completeness of Java\(^b\) programs.

Java\(^b\) programs are well-formed under the same conditions as Java\(^a\) programs, i.e. if their descriptions are well-formed and all method bodies have a type compatible with the type declared in their headers.

Furthermore, a Java\(^b\) program is complete, i.e. \( \vdash B \diamond \), if it is well-formed and contains all the descriptions necessary to check it is well-formed, i.e. if \( D(B) \vdash B \diamond \). Note that well-formedness of Java\(^b\) programs can only be established in the context of descriptions which contain the descriptions of that program and possibly more descriptions. For example, \( D_B \vdash B \diamond \) and \( D_B \vdash B \diamond \). Moreover, completeness is strictly more than well-formedness, e.g. \( B \not\vdash B \diamond \), but \( B \not\vdash B \diamond \).

The following lemma says that in a complete Java\(^b\) program any class that widens to a superclass or superinterface provides an implementation for each method exported by the superclass or superinterface.

Lemma 6 For any Java\(^b\) program \( B \) with \( \vdash B \diamond \), normal types \( T, T_0, T_1, \ldots, T_n \), exception classes \( E_1, \ldots, E_q \), class \( C \), if

- \( B \vdash C \leq_w T_0 \),
- \( \text{methH} \in \text{Meths}(D(B), T_0, m) \) with \( \text{ArgT}(\text{methH}) = T_1 \times \ldots \times T_n \) and \( \text{RetT}(\text{methH}) = T \) and \( \text{ExcT}(\text{methH}) = ET \),
Figure 19: Well-formed and complete Java\(^b\) programs

\[
\begin{align*}
\vdash & (Stmnt |)^* \\
\text{Stmnt} & ::= \text{if Expr then Stmts else Stmts} \mid \text{Var = Expr} \\
& \quad \mid \text{Expr.[ArgType][MethodName(Expr*)} \mid \text{throw Expr} \\
& \quad \mid \text{try Stmts (catch ClassName Id Stmts)^* finally Stmts} \\
& \quad \mid \text{try Stmts (catch ClassName Id Stmts)^*} \\
\text{Expr} & ::= \text{Value} \mid \text{Var} \mid \text{Expr.[ArgType][MethodName(Expr*)} \\
& \quad \mid \text{new ClassName}() \mid \text{new SimpleType([Expr]^+)([])^*} \\
& \quad \mid \text{Stmts [return Expr] \mid this} \\
\text{Var} & ::= \text{Name} \mid \text{Expr.[ClassName][VarName] \mid Expr[Expr]} \\
& \quad \mid \text{RefValue.[ClassName][VarName] \mid RefValue[Expr]} \\
\text{Value} & ::= \text{PrimValue} \mid \text{RefValue} \\
\text{RefValue} & ::= t \mid \text{null} \\
\text{PrimValue} & ::= i | \text{charValue} | \text{boolValue} | ... \\
\text{SimpleType} & ::= \text{PrimType} | \text{ClassName} | \text{interfaceName} \\
\text{PrimType} & ::= \text{bool} | \text{char} | \text{int} | ...
\end{align*}
\]

Figure 20: Java\(^r\) terms

10 Java\(^r\), the run-time language

As we said in the previous section, Java\(^b\) is an enriched version of Java\(^a\), extended with compile-time type information necessary at run-time. Moreover, new terms may appear at run time, the syntax of which is not covered by Java\(^b\). Therefore, we extend Java\(^b\) term syntax further to obtain Java\(^r\) term syntax – the run-time language. Java\(^r\), defined in figure 20, is a pure superset of Java\(^b\) term syntax.
The additional artifacts, which may occur at run-time and are not part of Java\(^b\), arise through addresses, the null value, and statements as expressions. Addresses have the form \(i, i', i_1, \ldots\); they represent references to objects and arrays, and may appear wherever a value is expected, as well as in array access and field access. Therefore, Java\(^f\) variables may have the form \(i, [\text{ClassName}]\text{VarName}\) or \(i[\text{Expr}]\), and expressions may have the form \(i\). An access to \(null\) may arise during evaluation of array access or field access; therefore, Java\(^f\) expressions may have the form \(null, [\text{ClassName}]\text{VarName}\) or \(null[\text{Expr}]\). Furthermore, in order to describe method evaluation through in-line expansion rather than closures and stacks, we allow an expression in Java\(^f\) to consist of a sequence of statements possibly terminated by a \text{return} statement, so that in the operational semantics a method call can be rewritten to such a statement sequence.

In order to describe the operational semantics of exception propagation and typing of terms containing \text{throw} \(a\) succinctly, we introduce the notion of context. The context of an exception, defined in figure 21, encompasses all enclosing terms up to and excluding the nearest enclosing \text{try-catch} \(a\) close, i.e. up to the first possible position at which the exception might be handled.

## 11 The operational semantics

Figure 22 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\(^f\) terms and states, or just states. The operational semantics, \(\sim\), is a mapping from Java\(^b\) programs and configurations to configurations.

| Configuration | ::= | Java\(^f\) term \times State \cup State |
| \(\sim\) | ::= | Java\(^b\) program \rightarrow Configuration \rightarrow Configuration |
| \(\gamma\) | ::= | Configuration \rightarrow Configuration |
| State | ::= | \{Ident \rightarrow \text{Value}\}^* \cup \{\text{RefValue} \rightarrow \text{ObjectOrArray}\}^* |
| ObjectOrArray | ::= | Object \mid \text{Array} |
| Object | ::= | \llbracket (\text{FieldName} : \text{ClassName} \rightarrow \text{Value})^*, \text{ClassName} \rrbracket |
| Array | ::= | \llbracket (\text{Value})^\ast, \text{ArmyType} \rrbracket |

Figure 22: Java\(^f\) run-time model
Each field also carries the class in which it was defined; this serves to distinguish fields of same name but declared in different classes, c.f. [14], ch. 15.10.1. For the philosophers example from section 2.1, \( \llike \Phi i_1 \), \( \llike \Phi i_2 \), \( \llike \Phi i_3 \), \( \llike \Phi i_4 \) is an object of class \( \Phi i_1 \). It inherits the field \( \llike \Phi i_1 \) from \( i_1 \), and has the field \( \llike \Phi i_1 \) from \( i_2 \).

The following state \( \sigma_0 \) contains mappings according to the philosophers example:

\[
\begin{align*}
\sigma_0(i_1) &= i_1 \\
\sigma_0(i_2) &= i_2 \\
\sigma_0(i_3) &= i_3 \\
\sigma_0(i_4) &= i_4 \\
\end{align*}
\]

Arrays carry their dimension and type information, and they consist of a sequence of values for the first dimension. For example, \([3, 5, 8, 13]^\mathbb{N}\) is a one dimensional array of integers, and \([null, null, i_1]^{\Phi i_1}\) is a one dimensional array of \( \Phi i_1 \).

11.1 State and 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 in the program and environment, a requirement introduced in definition 24, section 13.1.

**Definition 18** For integers \( n, i \) with \( 1 \leq i \leq n \), object \( \text{obj} = \llike f_1 C_1 : \text{val}_1, \ldots, f_i C_i : \text{val}_i, \ldots, f_n C_n : \text{val}_n \rrightarrow C^\prime \), state \( \sigma \), value \( \text{val} \), reference \( i \), identifier or address \( z \), class \( C \), field identifier \( f \), integers \( m, r, k, j \) with \( m \geq 0 \), \( 0 \leq k \leq r - 1 \), \( 0 \leq j \leq r - 1 \), array \( \text{arr} = [\text{val}_0, \ldots, \text{val}_{r-1}]^m \), we define:

- the access to field \( f \) declared in class \( C \):

\[
\text{obj}(f, C) = \text{val}_i \quad \text{if} \quad f = f_i \quad \text{and} \quad C = C_i
\]

- the access to component \( f, C \) of an object stored at reference \( i \), in state \( \sigma \):

\[
\sigma(i, f, C) = \sigma(i)(f, C)
\]

- the access to the \( k \)-th component of \( \text{arr} \):

\[
\text{arr}[k] = \text{val}_k
\]

- a new state, \( \sigma' = \sigma[z \rightarrow \text{val}] \), such that:

\[
\begin{align*}
\sigma'(z) &= \text{val} \\
\sigma'(z') &= \sigma(z') \quad \text{for} \quad z' \neq z
\end{align*}
\]

- a new object, \( \text{obj}' = \text{obj}[f, C \rightarrow \text{val}] \), a new state, \( \sigma' = \sigma[j, f, C \rightarrow \text{val}] \):

\[
\begin{align*}
\text{obj}'(f, C) &= \text{val} \\
\text{obj}'(f', C') &= \text{obj}(f', C') \quad \text{if} \quad f \neq f' \quad \text{or} \quad C \neq C' \\
\sigma' &= \sigma[i \rightarrow \sigma(i)[f, C \rightarrow \text{val}]]
\end{align*}
\]

- a new array, \( \text{arr}' = \text{arr}[k \rightarrow \text{val}] \), and a new state, \( \sigma' = \sigma[i, k \rightarrow \text{val}] \):

\[
\begin{align*}
\text{arr}'[k] &= \text{val} \\
\text{arr}'[j] &= \text{arr}[j] \quad \text{if} \quad j \neq k \\
\sigma' &= \sigma[i \rightarrow \sigma(i)[k \rightarrow \text{val}]]
\end{align*}
\]

We distinguish ground terms, which cannot be further rewritten, and 1-ground terms, which are “almost ground” and may not be further rewritten if they appear on the left hand side of an assignment.
Definition 19 A Java term $t$ is

- value iff $t$ is a primitive value, or $t = \text{null}$, or $t = i$;
- ground iff $t$ is a value, or $t = \text{throw } i$;
- 1-ground iff $t = \text{id}$, or $t = i[C][f]$, or $t = \text{null}[C][f]$, or $t = i[k]$, or $t = \text{null}[k]$ for some identifier $\text{id}$, class $C$, field $f$, address $i$, integer $k$.

11.2 Execution

Figures 23, 24, 25, 26, and 27 describe the rewriting of Java terms. We chose small step semantics because this is 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 assignment to array components and the other assignments statements [27]. On the other hand, only small step semantics allows the description of co-routines [19].

Program execution requires type information occasionally, e.g. that about the fields of a class for object creation, or the subclass relationship for assignment to array components, etc. Therefore we use definitions 12, 13, 14 from section 6.4.

Expressions

In figure 23 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 25 prevent an expression like $x$, or $i[C][f]$, appearing on the left hand side of an assignment from being further rewritten. They allow an expression of the form $w[C][x][C][y][C][z]$ to be rewritten to an expression of the form $i[C][z]$. Furthermore, there is no rule of the form $i, \sigma \vdash \sigma(i), \sigma$. This is because there is no explicit dereferencing operator in Java. Objects are passed as references, and they are dereferenced only implicitly, when their fields are accessed.

Array access as described here adheres to the rules in ch. 15.12 of [14], which require full evaluation of the expression to the left of the brackets. Thus, with our operational semantics, the term $a[a := b][3]$ corresponds to the term $a[b][3]; a := b$.

The last six rules in figure 23 describe the creation of new objects or arrays, c.f. ch. 15.8-15.9 of [14]. Essentially, a new value of the appropriate array or class type is created, and its address is returned. The fields of the object and the components of the array are assigned initial values of the type to which they belong. Initial values are defined in ch. 4.5.5. of [14] and here in the following definition.

Definition 20 The initial value for a simple type $T$ defined in a program $P$ is:

$$\text{InitVal}(P, T) = \begin{cases} 
0 & \text{iff } T = \text{int} \\
\text{false} & \text{iff } T = \text{char} \\
\text{false} & \text{iff } T = \text{bool} \\
\text{null} & \text{iff } P \vdash T \subseteq T \text{ or } P \vdash T \leq T
\end{cases}$$

For example, for any program $P$ and a state $\sigma_0$, the expression $\text{new int}[2][3]$ would be executed as $\text{new int}[2][3], \sigma_0, \gamma, t_1, t_2$ where $\sigma_0(\langle 1 \rangle) = \langle 0, 0, 0, \text{int} \rangle$, $\sigma_0(\langle 2 \rangle) = \langle 0, 0, 0, \text{int} \rangle$, $\sigma_0(\langle 3 \rangle) = [t_1, t_2]^{\text{int}[3]}$.

Also, the expression $\text{new int}[2][3][4]$ would be executed as: $\text{new int}[2][3][4], \sigma_0, \gamma, t_1, t_2$ where $\langle 1 \rangle$ and $\langle 2 \rangle$ are new in $\sigma_0$, and have the following contents in $\sigma_{01}$:

$$\sigma_{01}(\langle 1 \rangle) = \langle \text{null}, \text{null}, \text{null}, \text{int}[3] \rangle$$
$$\sigma_{01}(\langle 2 \rangle) = \langle \text{null}, \text{null}, \text{null}, \text{int}[3] \rangle$$
$$\sigma_{01}(\langle 3 \rangle) = \langle t_1, t_2 \rangle^{\text{int}[3][4]}$$
Figure 23. Expression execution
**Statements**

Figure 24 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 statement returning an expression evaluates this expression until ground and replaces itself by this ground value – thus modeling methods returning values.

\[
\begin{array}{l}
\text{states} \cdot \sigma \gamma \cdot \sigma' \quad \text{states} \cdot \sigma \gamma \cdot \text{states}', \sigma' \\
\text{states;}\text{stat}, \sigma \gamma \cdot \text{states}, \sigma' \\
\text{stats;}\text{stat}, \sigma \gamma \cdot \text{states}, \sigma' \\
\text{if} \text{true then stats else stats',} \sigma \\
\text{if} \text{false then stats else stats',} \sigma \\
ev, \sigma \gamma \cdot \text{e',} \sigma' \\
\text{return e, } \sigma \gamma \cdot \text{return e',} \sigma' \\
\end{array}
\]

Figure 24: Statement execution

**Assignment**

Figure 25 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 modifies the state accordingly.

\[
\begin{array}{l}
v \text{is not l-ground} \\
v, \sigma \gamma \cdot \text{v',} \sigma' \\
v \text{is l-ground} \\
v = e, \sigma \gamma \cdot v = e', \sigma' \\
\text{val is value, k is integer value} \\
\text{id is an identifier} \\
\text{id=}\text{val, } \sigma \gamma \cdot \text{id=}\text{val1} \\
\text{let;} [\text{f=}\text{val, } \sigma \gamma \cdot \text{let;} \text{f, C=}\text{val}] \\
\text{null;} [\text{let;} \text{f=}\text{val, } \sigma \gamma \cdot \text{throw new NullPE(),} \sigma] \\
\text{null;} [\text{k=}\text{val, } \sigma \gamma \cdot \text{throw new NullPE(),} \sigma] \\
\text{val is value, k is integer value} \\
\text{id=}\text{val, } \sigma \gamma \cdot \text{id=}\text{val1} \\
\text{let;} [\text{f=}\text{val, } \sigma \gamma \cdot \text{let;} \text{f, C=}\text{val}] \\
\text{null;} [\text{let;} \text{f=}\text{val, } \sigma \gamma \cdot \text{throw new NullPE(),} \sigma] \\
\text{null;} [\text{k=}\text{val, } \sigma \gamma \cdot \text{throw new NullPE(),} \sigma] \\
\end{array}
\]

Figure 25: Assignment execution
The remaining rules describe assignments to fields and to array components. Firstly they check whether the object or array address is not null; if it is null then NullPE is thrown. Secondly, they check if the array index is within the array bounds; if not, IndexOutOfBoundsException is thrown. Thirdly, they check whether the value fits the array type; if not, ArrayStoreException is thrown. 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 the definition 21.

**Definition 21** A value val fits a type T in a program P, iff

- \( \vdash T \triangleleft_{\text{primAr}} \), or
- \( \not\forall T \triangleleft_{\text{primAr}} \) and \( T = T' \) and
  - \( \text{val} = \text{null} \), or
  - \( \sigma(\text{val}) = \ll \ldots \rr^C \) and \( P \vdash C \subseteq T' \), or
  - \( \sigma(\text{val}) = \ll \ldots \rr^D \) and \( P \vdash T' \leq T' \)

where \( \vdash T \triangleleft_{\text{primAr}} \) iff \( T = \text{int} \) or \( T = \text{char} \) or \( T = \text{bool} \)

Note that any value fits any array of primitive type, e.g. null fits type \text{int}[]]. This is so, because when values are assigned to components of arrays of primitive types, no run-time check needs to be performed, c.f. the type rules for assignment in Java in figure 28.

Finally, if these requirements are satisfied, then the assignment is performed. In case an exception is thrown it is propagated by the exception rules from figure 27.

Note also that we have no rule of the form \( \iota := \text{value, } \sigma' \ldots \). This is because in Java overwriting of objects is not possible – only sending messages to them, or overwriting selected instance variables.

**Method call**

Figure 26 describes the evaluation of method calls. The receiver and argument expressions are evaluated left to right, e.g. ch. 9.3 in [14]. The first rule describes rewriting of the \( k^{th} \) expression, where all the previous expressions (i.e. \( \text{val}_i, 1 \leq i \leq k - 1 \)) 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 \( t'[/x/ \) has the usual meaning of replacing \( x \) by the term \( t' \) in the term \( t \).

Execution of the method call \( \text{aPhil}.[\text{Quest}].\text{think(aQuest)} \) in the state \( \sigma_0 \) described in section 11 results in the rewrites:

\[ \begin{align*}
\text{aPhil}.[\text{Quest}].\text{think(aQuest)}, & \quad \sigma_0 \triangleleft_\alpha \quad \iota_1, [\text{Quest}].\text{think(aQuest)}, \sigma_0 \triangleleft_\beta \\
\iota_1, [\text{Quest}].\text{think(\iota_4)}, & \quad \sigma_0 \triangleleft_\alpha \quad (z_0, [\text{FrPhil}] \text{like=oyster;...}), \sigma_1 \triangleleft_\beta \\
(\ldots), & \quad \sigma_2
\end{align*} \]

where \( \sigma_{ph} \) was introduced in section 7, and the states \( \sigma_1, \sigma_2 \) are:

\[ \begin{align*}
\sigma_1 = \sigma_0 \ll \ld \iota_4 \ll \ld \iota_4], \\
\sigma_2 = \sigma_1 \ll \ld \ll \text{Like Phil: } \iota_2, \text{like FrPhil: } \iota_3 \gg \text{FrPhil}]
\]

**Exceptions**

Figure 27 describes the operational semantics related to exceptions. Execution of \text{try-catch} and \text{try-catch-finally} statements proceeds by execution of the \text{try} part. If no exception is thrown in the \text{try} part, then the most enclosing \text{try-catch} statement terminates, whereas the most enclosing \text{try-catch-finally} statement proceeds to execute the \text{finally} part.

A \text{throw} statement evaluates the corresponding expression – note that evaluation of that expression might itself throw a further expression. If the expression evaluates to null, then a NullPE is thrown. Once the expression following \text{throw} is ground, i.e. once it reaches the form \( \iota, \) the exception is propagated.
val\textsubscript{i} is value for i ∈ {0..k - 1}, n ≥ k ≥ 0
\begin{align*}
\sigma_\text{val}_i, \sigma' &\vdash \text{val}_i, [\text{AT} = (\text{val}_1, \text{val}_2, \ldots, \text{val}_k), \sigma_\gamma] \\
\text{null}, [\text{AT} = (\text{val}_1, \text{val}_2, \ldots, \text{val}_k), \sigma_\gamma] &\vdash \text{throw new NullPE}, \sigma' \end{align*}
\begin{align*}
\text{val}_i \text{ is value for } i \in \{0..n\}, \ n \geq 1 \\
\sigma(\text{val}_0) = \langle \ldots, ^\text{\texttt{null}} \rangle \\
\text{FirstFit}(P, n, C, \text{AT}) = \text{meth} \\
\text{meth} = T = (T_1, p_1, \ldots, T_n, p_n) \text{ throws } E_1, \ldots, E_q \{ \text{mBody} \} \\
z_i \text{ are new identifiers in } \sigma, \ i \in \{0..n\} \\
\sigma' = \sigma[z_0 \mapsto \text{val}_0] \ldots [z_n \mapsto \text{val}_n] \\
\text{mBody}' = \text{mBody}[z_0 / \text{this}, z_1 / p_1, \ldots, z_n / p_n] \\
\text{val}_0, [\text{AT} = (\text{val}_1, \text{val}_2), \sigma, \gamma' \vdash \text{mBody}', \sigma']
\end{align*}

Figure 26: Evaluation of method call

Thus, the exception reaches either the outer level of the program, or it reaches an enclosing \texttt{try-catch} or \texttt{try-catch-finally} statement. An exception of a class not covered by any of the classes in the \texttt{catch} clauses is propagated out of the enclosing \texttt{try-catch} statement. Also, an exception of a class not covered by any of the classes in the \texttt{catch} clauses causes execution of the \texttt{finally} part of an enclosing \texttt{try-catch-finally} statement followed by renewed throwing of the original exception.

According to the last rule, an exception is handled by the first handler whose class is a superclass of the exception’s class: the statements of the corresponding \texttt{catch} clause are executed with the \texttt{catch} clause variable (v\textsubscript{i}) pointing to the exception object. The \texttt{finally} clause, if any, will follow execution of the handler independently of whether the handler executed normally or abnormally.

The last two rules in figure 27 reflect the contents of pages 291-294 of the Java\textsuperscript{b} worries example B\textsubscript{wr} in a state \(\sigma_{10}\), where
\begin{align*}
\sigma_{10}(\text{Peter}) &= t_{11} \\
\sigma_{10}(t_{11}) &= \langle \text{age Person}: 0 \rangle_{\text{Person}}
\end{align*}
Disregarding the possibility for unchecked exceptions like null dereferencing \texttt{etc.}, there are three possible outcomes, namely:
\begin{align*}
\sigma_{11} &= \text{no exception thrown} \\
\sigma_{14} &= \text{an \texttt{Illness} exception thrown and caught} \\
\text{throw } t_{13}, \sigma_{13} &= \text{a \texttt{Worry} exception thrown.}
\end{align*}
where the states are:
\begin{align*}
\sigma_{11} &= \sigma_{10}[t_{11} \mapsto \langle \text{age Person}: 1 \rangle_{\text{Person}}] \\
\sigma_{12} &= \sigma_{10}[t_{12} \mapsto \langle \text{severity Illness}: 0 \rangle_{\text{Illness}}] \\
\sigma_{13} &= \sigma_{12}[t_{13} \mapsto \langle \text{Worries} \rangle] \\
\sigma_{14} &= \sigma_{12}[t_{12} \mapsto \langle \text{severity Illness}: -10 \rangle_{\text{Illness}}]
\end{align*}
\[
\begin{align*}
\text{try } & \text{stats } E_1 \text{ v_1 stats_i ... catch E_n v_n stats_{\sigma} } \\
\gamma' & \text{try stats' } \text{catch E}_1 \text{ v_1 stats_i ... catch E_n v_n stats_{\sigma'} } \\
\text{try } & \text{stats } E_1 \text{ v_1 stats_i ... catch E_n v_n stats_{\sigma}} \text{ finally stats_{\sigma_{i+1}} } \\
\gamma' & \text{try stats' } \text{catch E}_1 \text{ v_1 stats_i ... catch E_n v_n stats_{\sigma}} \text{ finally stats_{\sigma_{i+1}} } \\
\text{stats, } & \sigma_{\gamma' \sigma'} \\
\text{try } & \text{stats } E_1 \text{ v_1 stats_i ... catch E_n v_n stats_{\sigma}} \text{\gamma'\sigma' } \\
\text{try } & \text{stats } E_1 \text{ v_1 stats_i ... catch E_n v_n stats_{\sigma}} \text{\gamma'\sigma' } \\
\text{e, } & \sigma_{\gamma' \sigma'} e', \sigma' \\
\text{throw } & \text{e, } \sigma \gamma' \text{throw } e', \sigma' \\
\text{throw null, } & \sigma \gamma' \text{throw new NullPE() } , \sigma \\
\text{cont}[\cdot ] \square \text{ a context} \\
\text{cont}[\cdot ] \text{throw } i, \sigma \gamma' \text{throw } i, \sigma \\
\sigma(i) = \ll ... \gg^{k} \\
\forall k \in \{1...n\} \ P \parallel E \subseteq E_k \\
\text{try throw } i \text{ catch E}_1 \text{ v_1 stats_i ... catch E_n v_n stats_{\sigma}} \text{\gamma'\sigma' } \\
\text{try throw } i \text{ catch E}_1 \text{ v_1 stats_i ... catch E_n v_n stats_{\sigma}} \text{ finally stats_{\sigma_{i+1}} } \\
\sigma(i) = \ll ... \gg^{k} \\
\exists i \in \{1...n\} : \quad P \parallel E \subseteq E_i \quad \text{AND} \quad \forall k \in \{1...i-1\} \ P \parallel E \subseteq E_k \\
\text{stats}' = \text{stats}_{i}[x/v_1] \quad z \text{ new in stats}_i \text{ and in } \sigma \\
\sigma' = \sigma[z \mapsto i] \\
\text{try throw } i \text{ catch E}_1 \text{ v_1 stats_i ... catch E_n v_n stats_{\sigma}} \text{\gamma'\sigma' } \\
\text{try throw } i \text{ catch E}_1 \text{ v_1 stats_i ... catch E_n v_n stats_{\sigma}} \text{ finally stats_{\sigma_{i+1}} } \\
\gamma' \text{try stats' finally stats_{\sigma_{i+1}} } \\
\end{align*}
\]

Figure 27: Exception throwing, propagation and handling
The following table demonstrates execution of the term \texttt{peter.[][act()} in more detail. We group consecutive executions in marked blocks. At the end of block 1, depending on the outcome of the condition, execution may continue at block 2 or 3. Similarly, at the end of block 3 execution may continue at block 4 or 5.

\begin{tabular}{|c|c|c|}
\hline
\textbf{Block 1 – start} & \textbf{Block 2 – the condition returns false} & \textbf{Block 3 – the condition returns true} \\
\hline
\texttt{peter.[][act()}. & \texttt{i_{111}.[][act()}. & \texttt{try \{} \texttt{if (...) throw diagnose().[]treat(); else age=age+1; } \\
\texttt{catch(Illness i){ i.[]cure(); }}. & \texttt{catch(Illness i){ i.[]cure(); }}. & \texttt{try \{} \texttt{throw i_{12}}().[]treat(); \\
\texttt{catch(Illness i){ i.[]cure(); }}. & \texttt{catch(Illness i){ i.[]cure(); }}. & \texttt{catch(Illness i){ i.[]cure(); }}. \\
\texttt{continue at 2} or 3 \texttt{}}. & \texttt{continue at 2} or 3 \texttt{}}. & \texttt{continue at 2} or 3 \texttt{}}. \\
\hline
\textbf{Block 4 – the condition returns false} & \textbf{Block 5 – the condition returns true} & \\
\hline
\texttt{try \{} \texttt{throw this;} \texttt{\}}. & \texttt{try \{} \texttt{throw (throw new Worry());} & \\
\texttt{catch(Illness i){ i.[]cure(); }}. & \texttt{catch(Illness i){ i.[]cure(); }} & \texttt{throw i_{13}, \sigma_{13}} \\
\texttt{continue at 4} or 5 \texttt{}}. & \texttt{catch(Illness i){ i.[]cure(); }}. & \\
\hline
\end{tabular}

12 The Java\textsuperscript{r} type system

We give types to Java\textsuperscript{r} terms in order to be able to formulate a subject reduction theorem.

The type of an address \texttt{t} depends on the object or array pointed at in the current state \(\sigma\). Therefore, the type of a Java\textsuperscript{r} term depends not only on the program and environment, but also on the state. That is why type judgements for Java\textsuperscript{r} terms \texttt{t} have the forms \(D, V, \sigma_{i} : T\) and \(D, V, \sigma_{i} : T : E\), giving the normal type \(T\), and the abnormal type \(ET\). The type rules for the Java\textsuperscript{r} normal types are given in figure 28, the type rules for the Java\textsuperscript{r} abnormal types are given in figure 29.

The Java\textsuperscript{r} type rules correspond to the Java\textsuperscript{b} type rules, except where Java\textsuperscript{r} introduces new syntax, or, where technicalities of the subject reduction theorem proof require otherwise. There are eight cases where Java\textsuperscript{r} types differ from Java\textsuperscript{b} types. The reasons for the differences can be classified into two categories.

Firstly, those that give types to expressions that may arise only during program execution. For example, addresses may appear in Java\textsuperscript{r} expressions, and a term \texttt{throw t} may be propagated and become a subterm of a method call, field access, etc.
Secondly, those that give types to terms that would be type-incorrect in Java \(^a\) (i.e. typing of assignments, the rules for null and for field access in figure 28). The rules in this category make type-correct terms which would have been type-incorrect in Java \(^a\). However, the evaluation of such terms does not corrupt the integrity of the system, since unless infinite, it will throw an appropriate exception.

Below we discuss these categories in more detail. Figure 28 describes the normal types for Java \(^b\) terms.

There are two rules from the first category. They describe the types of addresses and can be found in the fifth and sixth groups of rules in figure 28. If an object is stored at address \(i\), i.e. \(\sigma(i) = \ll\ldots\rr\), then its class, \(C\), is the type of \(i\). If a \(k\)-dimensional array of \(T\) is stored at such an address, i.e. \(\sigma(i) = \ll\ldots[\cdot]\rr\), then \(T[\cdot][\cdot]\ldots[\cdot]\) is the type of \(i\).

The rule describing the normal type of null belongs to the second category. It says that null has any reference type. This rule is required in order to give a type to terms like \(\text{null}[j-4]\), which, although type-incorrect in Java \(^a\), may arise during execution of Java \(^b\) terms. Such terms ultimately lead to the \texttt{Null1PE} exception, but not immediately, because the Java semantics requires other parts of the expression to be evaluated first; in our example, \(j-4\) has to be evaluated first and might throw another exception. In order to be able to prove the subject reduction theorem, such expressions need a type. The effect of this rule is that Java \(^a\) terms do not have unique types.

The rule for field access in Java \(^b\) is different from those in Java \(^a\) and Java \(^b\). The difference is best explained using the philosophers example. The Java \(^a\) term \(\text{aPhil}(|\text{Phil}||\text{like})\) has the type \(\text{Truth}\), whereas the Java \(^b\) term \(\text{pascal}(|\text{Phil}||\text{like})\) is type-incorrect. Nevertheless, execution of \(\text{aPhil}(|\text{Phil}||\text{like})\) in the state \(\sigma\) may lead to the configuration \(\sigma(|\text{Phil}||\text{like})\), \(\sigma\) where \(i\) points to a FrPhil object, i.e. \(\sigma(i) = \text{aPhil} = \) \(i\) and \(\sigma(i) = \ll\ldots\rr\) \text{FrPhil}. Obviously, the term \(\sigma(|\text{Phil}||\text{like})\) is type-correct. Therefore, the Java \(^b\) requirement that the descriptor \(C\) is the class of the first visible field has to be weakened to the Java \(^a\) requirement that the descriptor denotes a superclass which contains a definition of such a field, e.g. \(\text{Phil}||\text{Truth}\) \(\notin\) FirstVis\(\langle\text{pascal},\text{FrPhil},\text{like}\rangle\) but \(\langle\text{Phil}||\text{Truth}||\text{like}\rangle \in \text{Fields(}\text{pascal},\text{FrPhil},\text{like})\).

The rules describing assignments belong to the second category. Assignment to variables which are not array components (i.e. where \(v\) does not have the form \(\text{a[e]}||\)) or to array components of primitive types are type-correct only if the left-hand side has a type that widens to that of the right-hand side. This is described by the first two array assignment rules and is the same as in the Java \(^a\) system. However, assignments to array components are type-correct if the left-hand side and the right-hand side are type-correct, and the array is an array of reference type, i.e. is not a primitive array. Thus, the terms \(i[2]=\text{aPhil}\), \(i[2]=17\) are type-correct even if \(i\) points to an array of FrPhil. Obviously, the corresponding assignment in Java \(^a\), e.g. \(\text{dining}[2]=\text{aPhil}\) with \(\text{dining}\) as an array of FrPhil is type-incorrect. The reason for the weaker type requirement in Java \(^a\) comes from the fact that the validity of the assignment can not be determined before the right-hand side is fully evaluated. Namely, if \(\text{aPhil}\) returns an object which does not belong to FrPhil or any of its subclasses, then the exception \texttt{ArrStoreE} will be thrown. Therefore, the intermediate term \(i[2]=\text{aPhil}\) has to be considered type-correct in Java \(^b\). Interestingly, such a distinction between types for array assignments and other assignments is not necessary when using large steps operational semantics [27].

The rule giving types to terms \(\text{cont} \subseteq t \supseteq \langle T || T\rangle\) which contain a subterm \(t\) of type \(\langle T || T\rangle\) belong to the second category too. Such terms may only arise through exception throwing and propagation.

Figure 29 describes the abnormal types for Java \(^b\) terms. The type rules are similar to the rules describing abnormal types for Java \(^a\) and Java \(^b\) terms in figures 12, 18 except the rules concerned with addresses (which are not part of the Java \(^b\) syntax). Therefore, figure 29 contains one additional rule which determines that the abnormal type of an address is empty.

12.1 Properties of the Java\(^b\) type system

Definition 22 \(D, V, T, e \rightarrow T \leftrightarrow E\) \(\iff D, V, T, e \rightarrow t \leftrightarrow E\) and \(D, V, T, e \rightarrow t : E\)

Any well-typed Java\(^b\) expression retains its type for any state \(\sigma\).

50
i is integer, c is character, D ⊨ V, V(x) = T x
D ⊨ T ≤w Object
D, V, σ|_n^t true : bool, D, V, σ|_n^t false : bool,
D, V, σ|_n^t i : int, D, V, σ|_n^t c : char, D, V, σ|_n^t x : T

D, V, σ|_n^t e : bool
D, V, σ|_n^t stats : T D, V, σ|_n^t stats' : T' D, V, σ|_n^t stat : T'/*
D, V, σ|_n^t if e then stats else stats' : T ∪ T'

v ≠ e'[e''] for any e', e''
D, V, σ|_n^t v : T
D, V, σ|_n^t e[e''] : T
D, V, σ|_n^t e' : T'
D, V, σ|_n^t e[e'''] : T'
T' ⊆w T
D, V, σ|_n^t v = e : void
D, V, σ|_n^t e[e''] = e' : void
D, V, σ|_n^t e[e'''] = e' : void
T ≠ ⊥
D, V, σ|_n^t e : T
D, V, σ|_n^t return e : T

D ⊨ C ⊆ C
D, V, σ|_n^t new C(0) : C
D, V, σ|_n^t new C(i) : C

n ≥ 1, k ≥ 0
D ⊨ T ⊆vType
D, V, σ|_n^t e1 : int i ∈ {1..n}
D, V, σ|_n^t new T[e1]|...|e2]|...|Ek : T[1]|...|En
σ(i) = [...]|i|...|n
D, V, σ|_n^t i : T[1]|...|En

D ⊨ T ⊆ u
D, V, σ|_n^t e : T
D ⊨ T ≤w C
D, V, σ|_n^t e'[e''] : int
(C T' f) ∈ Fields(D, T, f)
D, V, σ|_n^t e : C[f] : T'

D ⊨ T0 ⊆ u
D, V, σ|_n^t e1 : T_i i ∈ {0..n}, n ≥ 0
D ⊨ T_i ⊆w T_i i ∈ {1..n}
FirstFields(D, T0, T1 x ... x Tn) = {methH}
D, V, σ|_n^t e0[T1 x ... x Tn]σ(e1, ..., e_n) : REF(methH)

D ⊨ E ⊆ Throwable
D, V, σ|_n^t e : E
D, V, σ|_n^t throw e : ⊥
D, V, σ|_n^t e : T, ⊢ cont ⊆ C is a context
D, V, σ|_n^t t : ⊥
D, V, σ|_n^t cont ⊆ C ⊆ ⊥

n ≥ 0
D ⊨ E_i ⊆ Throwable i ∈ {1..n}
D ⊨ V ⊆
V(v_i) = Undefined i ∈ {1..n}
D, V, σ|_n^t E_i v_i, σ|_n^t statsa_i : T_i i ∈ {1..n}
D, V, σ|_n^t statsa_0 : T_0
D, V, σ|_n^t statsa_{i+1} : T_{i+1}
D, V, σ|_n^t try statsa_0 catch E_i v_i statsa_1 ... catch E_n v_n statsa_n : void
D, V, σ|_n^t try statsa_0 catch E_i v_i statsa_1 ... catch E_n v_n statsa_n
D, V, σ|_n^t finally statsa_{i+1} : T_{i+1}

Figure 28: Normal types for Java terms.
\[ \text{i is integer, c is character, } V(x) \neq \text{Undefined} \]
\[
\begin{align*}
D, V, \sigma x^0 & \text{ true : } \emptyset, \quad D, V, \sigma x^0 \text{ false : } \emptyset \\
D, V, \sigma x^1 i : \emptyset, \quad D, V, \sigma x^1 c : \emptyset, \quad D, V, \sigma x^1 x : \emptyset \\
D, V, \sigma x^2 e : ET \\
D, V, \sigma x^2 \text{ stats : } ET', \quad D, V, \sigma x^2 \text{ stats' : } ET'', \quad D, V, \sigma x^2 \text{ stat : } ET''' \\
D, V, \sigma x^2 \text{ stats ; stat : } ET \cup ET'' \\
D, V, \sigma x^2 \text{ if e then stats else stats' : } ET \cup ET'' \\
D, V, \sigma x^2 v : ET \\
D, V, \sigma x^2 e : ET' \\
D, V, \sigma x^2 v = e : ET \cup ET'' \\
D, V, \sigma x^2 \text{ return e : ET} \\
D, V, \sigma x^2 \text{ new C() : } \emptyset \\
D, V, \sigma x^2 e i : ET_i, \quad i \in \{1..n\} \\
D, V, \sigma x^2 \text{ new T[e1]...[en]} : ((\text{LET} i_{1..n}) \\
D, V, \sigma x^2 e : ET \\
D, V, \sigma x^2 e' : ET' \\
D, V, \sigma x^2 e[e'] : ET \cup ET'' \\
D, V, \sigma x^2 e i \{C\} : ET \\
D, T_0 \triangleright \theta \\
D, V, \sigma x^2 e_0 : T_0 \\
\text{FirstFilt}(D, m, T_0, T_1 \times ... \times T_n) = \{\text{methH}\} \\
D, V, \sigma x^2 e_1 : ET_i, \quad i \in \{0..n\} \\
D, V, \sigma x^2 e_0[T_1 \times ... \times T_n] \text{ e} : ((\text{LET} i_{1..n}) \cup \text{ErrF}(\text{methH}) \\
e \neq \text{null} \\
D, E \triangleright \text{ Throwable} \\
D, V, \sigma x^2 e : E \\
D, V, \sigma x^2 e : ET \\
D, V, \sigma x^2 \text{ throw null : } \text{NullPE} \\
D, V, \sigma x^2 \text{ cont } \square \square : ET \\
D, E \triangleright \text{ Throwable, i } \in \{1..n\} \\
D, v \triangleright \text{ Void} \\
\text{V(v1) = Undefined, i } \in \{1..n\} \\
D, V, \text{ E[v1, \sigma x^1 \text{ stats}_1 : ET}_i, \quad i \in \{1..n\} \\
D, V, \sigma x^1 \text{ stats}_0 : ET_0 \\
D, V, \sigma x^1 \text{ stats}_{n+1} : ET_{n+1} \\
D, V, \sigma x^2 \text{ try stats}_0 \text{ catch E[v1 stats1... catch E[v2 stats}_2 \text{ stats}_n :} \\
(ET_0 \cup \{E[v1, \ldots, E[v2, \ldots, ET_{n+1}] \cup \text{LET} i_{1..n} \\
D, V, \sigma x^2 \text{ try stats}_0 \text{ catch E[v1 stats1... catch E[v2 stats}_2 \text{ stats}_n \text{ finally stats}_{n+1} :} \\
(ET_0 \cup \{E[v1, \ldots, E[v2, \ldots, ET_{n+1}] \cup \text{LET} i_{1..n}
\]

Figure 29: Abnormal types for Java\textsuperscript{5} terms.
Lemma 7 For descriptions D, environment V, type T, T \neq \text{nil} \parallel ET, and Java\textsuperscript{a} term t:

\[ D, V \parallel t : T \implies \forall \text{ states } \sigma : D, V, \sigma \triangledown t : T \]

Notice, that the opposite direction does not hold. For the array example \( P_{\text{al}} \), for a variable dining of type \( \text{PrPhil}[] \), the Java\textsuperscript{a} term \( \text{dining}[2] = \text{aPhil} \) is type-correct, but the corresponding Java\textsuperscript{b} term, dining[2] = aPhil is not. Furthermore, Java\textsuperscript{a} expressions may have more than one type.

Lemma 8 \( D, V, \sigma \triangledown t : \bot \parallel ET \implies ET \neq \emptyset \)

Lemma 9 For any Java\textsuperscript{a} term t: D, V, \sigma \triangledown t : \bot \parallel ET' \iff

\begin{itemize}
  \item \( \exists E : D \vdash E \equiv \text{Throw} \text{a} \text{ble} \), and \( \exists ET : D \vdash ET \odot \text{AtnType} \) such that \( t = \text{throw } e \) and \( D, V, \sigma \triangledown e : E \parallel ET \) and \( ET' = \{E\} \cup ET \), or
  \item \( \exists \text{ context cont } \subseteq \cdot \subseteq \text{ and term } t' : t = \text{cont } \cap t' \subseteq \text{ and } D, V, \sigma \triangledown t' : \bot \parallel ET' \)
\end{itemize}

Definition 23 For a Java\textsuperscript{a} term t, program P, description D with D = D(P), environment V and state \( \sigma \) we define:

\[ P, V, \sigma \triangledown t : T \text{ if } D, V, \sigma \triangledown t : T; \]

\[ P, V, \sigma \triangledown t : T \text{ if } D, V, \sigma \triangledown t : T; \]

\[ P, V, \sigma \triangledown t : ET \text{ if } D, V, \sigma \triangledown t : ET. \]

13 Soundness of the Java type system

13.1 Conformance

In order for the execution of a well-typed term t to behave as expected, the state \( \sigma \) should conform to P and V, i.e. it should satisfy \( P, V \vdash \sigma \lhd \), a property described in definition 24. This requires all variables and addresses to contain values which conform to the variable type expected in V and P. Objects are required to be constructed according to their classes, array values are required to conform to their dimensions and to consist of values of appropriate variable types as defined in a program, and variables are required to contain values of the appropriate variable types as declared in the environment defined by the descriptions.

Conformance of a value \( \text{val} \) to a variable type T, i.e. \( P, \sigma \vdash \text{val} \lhd T \), is described in definition 24 in terms of the weaker, local conformance \( P, \sigma \vdash \text{val} \lhd T \), introduced for technical reasons, in order to obtain a well-founded relationship.

In definition 24 we also describe conformance for configurations: \( P, V \vdash t, \sigma \lhd T \), if the state \( \sigma \) conforms to P and V, and the term t has the type T. An environment V, that contains all variable declarations from another environment \( \nu \) and possibly some additional ones, is said to conform to \( \nu \), i.e. \( V \lhd \nu \). Similarly, a state \( \sigma \) may conform to another state \( \sigma' \), i.e. \( \sigma \lhd \sigma' \).

Definition 24 For a value \( \text{val} \), variable type T, program P, state \( \sigma \) and integer j we define:

\begin{itemize}
  \item \( P, \sigma \vdash \text{val} \lhd T \iff \)
  \begin{itemize}
    \item \( \text{val} \) is a primitive value, T is a primitive type and \( \text{val} \in \text{ET} \), or
    \item \( \text{val} = \text{null} \), and T is a class, interface or array type, or
    \item \( \text{val} = \nu, \sigma(i) = \llbracket \ldots \rrbracket^\nu \text{ and } P \vdash C \leq_w T \), or
    \item \( \text{val} = \nu, \sigma(i) = \llbracket T_1 \ldots T_k \rrbracket \leq_w T \)
  \end{itemize}
  \item \( P, \sigma \vdash \text{val} \lhd T \iff \)
\end{itemize}
\[\begin{align*}
- P, \sigma \vdash val \ll\ll T \quad \text{and} \\
\star \mathop{val=\iota} \quad \text{and} \quad \sigma(\iota) = \langle f_1, C_1 ; v_1, …; f_n, C_n ; v_n \rangle_c \implies \\
\forall \text{fields } f, \text{ classes } C', \text{ variable types } T', \text{ with } (C', T', f) \in \text{Fields}(P, C, f) \quad \\
\exists i \in \{1, …, n\} \text{ with } f_i = f, C_i = C' \quad P, \sigma \vdash val_k \ll\ll T' \quad \text{or} \\
\star \mathop{val=\iota} \quad \text{and} \quad \sigma(\iota) = \langle \text{val}_0, …; \text{val}_{n-1} \rangle_{T'}, \implies \\
\forall i \in \{0, …, n-1\} : P, \sigma \vdash val_i \ll\ll T', \ll\ll \cdots, \ll\ll \ll\ll k
\end{align*}\]

- \(P, \sigma \vdash \sigma \quad \text{iff} \quad P \vdash T \)

- \(P, \sigma \vdash v(x) \iff P, \sigma \vdash \sigma(\iota) < T, \quad \sigma(\iota) = \langle \ldots, \rangle_c \implies P, \sigma \vdash \iota < C, \quad \ldots\)

- \(P, \sigma \vdash t, \sigma \ll\ll T \quad \text{iff} \quad P, \nu \vdash t : T \)

- \(v \ll\ll v' \quad \text{iff} \quad v'(x) \neq \text{Undefined} \implies v(x) = v'(x) \)

- \(\sigma \ll\ll \sigma' \quad \text{iff} \quad \sigma'(x) \neq \text{Undefined} \implies \sigma(x) = \sigma'(x) \quad \text{and} \quad \ldots\)

For example, for the state \(\sigma_0\) from section 11, the program \(B_{ph}\), given in section 7, and the environment \(V_{ph} = \text{Phil aPhil; Quest aQuest; the following holds: } B_{ph}, V_{ph} \vdash \sigma_0 \quad \sigma\)

The fitting requirement from definition 21 is weaker than conformance of values. Both are defined on a program but conformance considers a value and a large part of a state, whereas fitting considers the particular value only.

The following lemma states that conforming environments and states preserve all properties.

**Lemma 10** For program \(P\), description \(D\), environments \(V, V'\) satisfying \(v \ll\ll v'\), states \(\sigma, \sigma'\) with \(\sigma \ll\ll \sigma'\), term \(t\), and type \(T\):

- \(D \vdash V \quad \text{iff} \quad D \vdash V' \quad \text{iff} \quad P \vdash V \quad \text{iff} \quad P \vdash V' \quad \text{iff} \quad P, V \vdash t : T \quad \text{iff} \quad P, V' \vdash t : T \quad \text{iff} \quad P, V, \sigma \vdash \sigma t : T \quad \text{iff} \quad P, V, \sigma' \vdash \sigma' t : T \)

**13.2 Properties of term evaluation**

The operational semantics is deterministic up to renaming of addresses and identifiers. Program execution may modify the contents of arrays and objects, but does not change their type or class.

**Lemma 11** For program \(P\), environment \(V\), state \(\sigma\) with \(P, V \vdash \sigma \quad \text{well-typed Java}^b\) term \(t\):

- \(t, \sigma \ll\ll t', \sigma' \quad \text{and} \quad t, \sigma \ll\ll t'', \sigma'' \quad \text{implies that} \quad t' = t'', \quad \sigma' = \sigma'' \quad \text{up to renaming of addresses and identifiers. Also,} \quad t, \sigma \ll\ll t', \sigma' \quad \text{and} \quad t, \sigma \ll\ll t'', \sigma'' \quad \text{implies that} \quad \sigma' = \sigma'' \quad \text{up to renaming of addresses and identifiers. Furthermore, it is impossible to have} \quad t, \sigma \ll\ll t', \sigma' \quad \text{and} \quad t, \sigma \ll\ll t'', \sigma'' \quad \text{and} \quad t, \sigma \ll\ll \sigma'\).
• If \( t, \sigma \models t', \sigma' \), then for any \( i \), if \( \sigma(i) = \ldots [ i] \ldots [ n] \) then \( \sigma'(i) = \ldots [ i] \ldots [ n] \), and if \( \sigma(i) = \ll \ldots \gg \) then \( \sigma'(i) = \ll \ldots \gg \).

The following lemma states that execution of a term \( t \) does not have any effect on the type of another term \( t'' \), even if one should be a subterm of the other.

**Lemma 12** For Java\(^F\) terms \( t, t', t'' \), states \( \sigma, \sigma' \), environment \( V \), types \( T, T'' \), Java\(^b\) program \( P \) if

- \( P, V, \sigma \not\vdash t : T \) and \( P, V, \sigma \not\vdash t'' : T'' \),
- \( P, V \vdash \sigma \checkmark \)
- \( t, \sigma \models t', \sigma' \) or \( t, \sigma \models t'', \sigma'' \),

then
- \( P, V, \sigma' \not\vdash t'' : T'' \).

The lemma may be surprising and is unusual. As stated later in the subject reduction theorem, a term \( t \) when rewritten to a new term \( t' \) has, possibly, a narrower type; therefore, one would expect evaluation of the term \( t \) to affect the type of a third term \( t'' \). However, according to the above lemma, even if \( t'' \) contained \( t \) as a subterm or \( t \) contained \( t'' \) as a subterm, the type of \( t'' \) would not change. The lemma is proven by structural induction over term execution (i.e., on \( t, \sigma, t', \sigma', t, \sigma, t'', \sigma'' \), and then, each case by structural induction on the typing of \( t'' \) (i.e., on \( P, V, \sigma \models t'' : T'' \)). The interesting cases are those where the state changes, i.e. the application of the three different assignment rules from figure 25. The normal type of \( t'' \) does not change, because it depends on static type information as well as on the run-time content of the store \( \sigma \). Assignments do not change the normal types of variables; these are looked up in the environment. They do not change the normal type of addresses (as shown in lemma 11). They do not change the normal type of field access \( e[C] \) because this depends on the type of the array \( e \) and on the type of the actual array component stored at \( e[C] \). Finally, they do not change the normal type of field access \( e[C] \) because this depends on the normal type of \( e \) and on the class \( C \) and does not depend on the value stored at \( e[C] \).

### 13.3 Subject reduction and soundness

The subject reduction theorem says that any non-ground well-typed Java\(^a\) term either terminates in a new state, or rewrites to another well-typed term of a type that can be widened to the type of the original term. Furthermore, the state remains consistent with the program and the environment. The subject reduction theorem of this paper is stronger than usual subject reduction theorems in the following three aspects.

Firstly, 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. Thus, for statically type-correct expressions it guarantees that the situation where an object cannot execute a message (the Smalltalk counterpart to “object does not understand message”) will never occur. In that sense it combines traditional the subject reduction and liveness properties.

Secondly, it guarantees that any exception thrown is of a class compatible with the abnormal type of the term.

Thirdly, although the theorem does not preclude the usual run-time errors like index out of bound, or erroneous assignment to array components, it does guarantee that such erroneous situations will raise a predefined unchecked exception, as opposed to going unnoticed and corrupting the run-time environment.

**Theorem 1 Subject Reduction** For Java\(^b\) program \( P \) with \( \vdash P \checkmark \), environment \( V \), state \( \sigma \), non-ground Java\(^F\) term \( t \), type \( T \) with \( P, V \vdash t, \sigma \models T \)

either
\[
\begin{align*}
&\exists \sigma', \nu', t', T' \text{ such that} \\
&\quad \text{t}, \nu_0, t', \sigma' \text{ and} \\
&\quad \nu \subseteq \nu, \quad \text{and} \\
&\quad P, \nu \vdash t', \sigma' \subseteq T' \text{ and } P \vdash T' \subseteq_w T,
\end{align*}
\]
or
\[
\begin{align*}
&\exists \sigma', \text{ET} : t, \sigma_0, \sigma' \text{ and } P, \nu \vdash \sigma' \text{ and } T = \text{void} \| \text{ET}.
\end{align*}
\]

Furthermore, if \( t \) is a non-\( l \)-ground variable and is not an array access, and \( T = T \| \text{ET} \), \( T' = T' \| \text{ET} \) and \( T' \neq \bot \), \( T \neq \bot \), then \( T = T' \).

The theorem is proven by structural induction over the derivations of \( P, \nu, \sigma \vdash t : T \).

**Proof**  Take \( D = D(P) \). As \( \not\vdash P \Delta \) then, by rules in figure 19, we also have
\[
\vdash D \Delta.
\]

By definitions 24, 23, figure 3, and definition 22, \( P, \nu \vdash t, \sigma \subseteq T \) implies that
\[
\begin{align*}
&\quad P, \nu \vdash \sigma \Delta \quad \text{(2)} \\
&\quad \text{and} \\
&\quad T = T \| \text{ET} \quad \text{and} \quad D, \nu, \sigma \vdash t : T, \quad D, \nu, \sigma \vdash t : \text{ET} \quad \text{(3)}
\end{align*}
\]

We prove the theorem by structural induction over the derivation of \( D, \nu, \sigma \vdash t : T \).

**Base case.**

1. **case.** \( t = \text{false} \) or \( t = \text{true} \), \( t = \text{i} \), \( t = \text{c} \), \( t = \text{x} \), where \( \text{i} \) is integer, \( \text{c} \) is character, \( \text{x} \) is identifier, is not applicable because \( t \) would be ground.

2. **case.** \( t = \text{\_} \) is not applicable either.

3. **case.** \( t = \text{new} \ C() \). To be completed.

**Induction step.** By figures 3, 7, we consider two cases for normal type \( T \) in (3).

1. **case.**

\[
T \neq \bot
\]

1.1 **case.** The last step in the normal typing of \( t \) was the method call rule from figure 28. Therefore, \( t \) has the form
\[
t = e_0.\{T_1 \times ... \times T_n\}(e_1, ..., e_n).
\]

So, the last step in the abnormal typing of \( t \) is the method call rule from figure 29. Hence, from figures 28, 29 we obtain:

\[
\begin{align*}
D \vdash T_0 \triangleq a, \quad &\text{(6)} \\
D, \nu, \sigma \not\vdash e_i : T'_i \| \text{ET}_i, \quad &\text{i} \in \{0...n\}, \quad n \geq 0, \quad \text{(7)} \\
D \vdash T'_i \subseteq_w T_i, \quad &\text{i} \in \{1...n\}, \quad \text{(8)} \\
\text{FirstFit}(D, \nu, T'_0, T_1 \times ... \times T_n) = \{\text{methH}\} \quad &\text{(9)}
\end{align*}
\]

Thus, we apply the method call type rule for \( t \) and obtain:
\[
D, \nu, \sigma \not\vdash t : \text{RetT(methH)} \| (\cup \text{ET}_i)_{i=0}^n \cup \text{ExcT(methH)}
\]

56
Therefore,

\[ T = \text{Rel}(\text{methH}) \parallel (\cup \text{ET}_i^\top_{i=0} \cup \text{Exc}(\text{methH})]. \]  

(11)

We define

\[ T_i = T_i^\top \parallel \text{ET}_i, \quad i \in \{0...n\}, n \geq 0. \]  

(12)

Because of (9), definition 15, definition of \( \text{Meth}(\cdot \cdot \cdot) \) in definition 5,

\[ T_0 \] is an interface or class type.

(13)

Therefore,

\[ T_i^\top \neq \bot, \]  

With (8) and widening definition in figure 8 we conclude that

\[ T_i^\top \neq \bot \quad \text{for} \quad i \in \{1...n\}. \]  

(15)

Hence, with (7), by (14) and (15),

\[ e_k \neq \text{throw} \ \iota \quad \text{for} \quad k \in \{0...n\} \quad \text{and address} \ \iota. \]  

(16)

1.1.1 case. The receiver \( e_0 \) is not ground. Therefore and because of (16), we apply the induction hypothesis and obtain cases 1.1.1.1 and 1.1.1.2.

1.1.1.1 case. \( \exists \sigma'', \nu'', e''_0, T''_0 \) such that:

\[ e_0, \nu'' \leq e''_0, \sigma'', \]  

\[ V'' \leq \nu. \]  

(17)

(18)

\[ P, \nu'' \vdash e''_0, \sigma'' \preceq T''_0 \ (i.e. \ P, \nu'', \sigma'' \vdash e''_0 : T''_0 \quad \text{and} \quad P, \nu'' \vdash \sigma'' \diamond), \]  

\[ P \vdash T''_0 \leq_w T_0. \]  

(19)

(20)

With (17) we apply the first rewrite rule for method call from figure 26:

\[ t, \nu'' \leq e''_0, [T_1 \times ... \times T_2]m(e_1,...,e_n), \sigma''. \]  

(21)

Define the term \( t' \), the state \( \sigma' \), and the environment \( \nu' \) as follows:

\[ t' = e''_0, [T_1 \times ... \times T_2]m(e_1,...,e_n), \]  

\[ \nu' = \nu''. \]  

(22)

(23)

\[ \sigma' = \sigma''. \]  

(24)

By figures 3, 7, there exist normal type \( T''_0 \), and abnormal type \( \text{ET}'_0 \) such that

\[ T''_0 = T''_0 \parallel \text{ET}'_0. \]  

(25)

Therefore, with (20) and (12), (13) and figure 7, by definition 13 and figure 8:

\[ D \vdash \text{ET}'_0 \subseteq \nu \text{ ET}_0, \quad \text{and} \]  

\[ D \vdash T''_0 \leq_w T'_0 \quad \text{or} \quad T''_0 = \bot \]  

(26)

1.1.1.1.1 case.

\[ D \vdash T''_0 \leq_w T'_0. \]  

(27)
Because of (9) and applying lemma 3, 2nd assertion, we obtain that

$$\exists \eta_{T_0}' \quad \text{with} \quad D \vdash T_0' \leq_w T_0'' \quad \text{and} \quad \text{meth} \in \text{MDec}(D, \eta_{T_0}'', \eta).$$

Applying lemma 3, 3rd assertion, with (27) we obtain that $$\exists \text{meth}'', \eta_{T_0}''$$:

$$\text{FirstFit}(D, \eta, \eta_{T_0}', T_1 \times \ldots \times T_n) = \{\text{meth}''\}, \quad \text{and} \quad (28)$$

$$D \vdash T_0'' \leq_w T_0''' \quad \text{and} \quad \text{meth}'' \in \text{MDec}(D, \eta_{T_0}''', \eta). \quad (29)$$

By lemma 1, we obtain that

$$\text{ArgT}(\text{meth}''') = \text{ArgT}(\text{meth}''), \quad (30)$$

$$\text{RetT}(\text{meth}''') = \text{RetT}(\text{meth}''), \quad (31)$$

Because of (7), (2), (17) and lemma 12 with (24):

$$D, V, \sigma' \models e_i : \; T_i' \parallel ET_i, \quad i \in \{1 \ldots n\}, \; n \geq 0. \quad (32)$$

With (23), (18) and lemma 10

$$D, V', \sigma' \not\models e_i : \; T_i' \parallel ET_i, \quad i \in \{1 \ldots n\}, \; n \geq 0. \quad (33)$$

With (23), (24), (19), (31), (8), (28), we apply the type rules for method call from figures 28, 29, and, by definition 22, we obtain the following type for $$t'$$:

$$D, V', \sigma' \not\models t' : \; \text{RetT}(\text{meth}''') \parallel ET''_0 \cup (\cup ET_i)_{i=1}^n \cup \text{ExcT}(\text{meth}''). \quad (34)$$

Define

$$T' = \text{RetT}(\text{meth}''') \parallel ET''_0 \cup (\cup ET_i)_{i=1}^n \cup \text{ExcT}(\text{meth}''). \quad (35)$$

With (11), (30), (31), (26), figure 8, and definition 6 we have proven that $$D \vdash T' \leq_w T$$, and with definition 13 that $$P \vdash T' \leq_w T$$.

Trivially, with (18), (19), (22), (23), (24), (34), definition 23, (35), by definition 24, $$P, V' \vdash t', \sigma' \triangleleft T'$$.

1.1.1.1.2 Case.

$$T''_0 = \bot. \quad (36)$$

Define a context cont $$\circ \; \circ$$ as follows:

$$\text{cont} \; \circ \; \circ = \circ \; \circ \; [T_1 \times \ldots \times T_n] \varepsilon(e_1, \ldots, e_k) \quad (37)$$

With (22), (37), we obtain that

$$t' = \text{cont} \; \circ \; \circ_{T''}. \quad (38)$$

Apply the type rule for context from figures 28, 29, and with (19), (25), (36), (23), (24), (38) obtain the following type for $$t'$$:

$$D, V', \sigma' \not\models t' : \; \bot \parallel ET''_0. \quad (39)$$

Define

$$T' = \bot \parallel ET''_0. \quad (40)$$

Because of (26), by definition of widening in figure 8, $$D \vdash T' \leq_w T$$, and, by definition 13, $$P \vdash T' \leq_w T$$.

Finally, with (18), (19), (22), (23), (24), (39), definition 23, (40), by definition 24, $$P, V' \vdash t', \sigma' \triangleleft T'$$.  

58
1.1.1.2 case. \( \exists \sigma_\mu, E_\nu : e_0, \sigma_\gamma \vdash \sigma_\nu, \) and \( P \vdash \sigma_\nu \downarrow. \) and \( T_0 = \text{void} \parallel E_\nu. \)

This case is not possible because of (13).

1.1.2 case. \( e_i \) is ground for \( i \in \{0, \ldots, k - 1\}, 1 < k < n, \) and \( e_k \) is not ground. Therefore and because of (16), by definition 19, we obtain that \( e_i \) values for \( i \in \{0, \ldots, k - 1\}. \) We apply the induction hypothesis and obtain cases 1.1.2.1 and 1.1.2.2.

1.1.2.1 case. \( \exists \sigma'_\nu, \nu', e'_\nu, T'_k \) such that:

\[
e_k, \sigma_\gamma \vdash e'_\nu, \sigma'_\nu.
\]

\( V' \subseteq V. \) \hspace{1cm} (41)

\( P, \nu' \vdash e'_\nu, \sigma'_\nu \leq T'_k \) (i.e., \( P, \nu', \sigma'_\nu \not\vdash e'_\nu : T'_k \); and \( P, \nu' \vdash \sigma'_\nu \downarrow), \) \hspace{1cm} (43)

\( P \vdash T'_k \leq_w T_k. \) \hspace{1cm} (44)

Therefore with (41), we apply the first rewrite rule for method call from figure 26:

\( t, \sigma_\gamma \vdash e_0, \nu, \tau_1 \times \ldots \times \tau_n \models (e_0, \ldots, e'_\nu, \ldots, e_k), \sigma'_\nu. \) \hspace{1cm} (45)

Define the term \( t', \) the state \( \sigma', \) and the environment \( V' \) as follows:

\( t' = e_0, \nu, \tau_1 \times \ldots \times \tau_n \models (e_0, \ldots, e'_\nu, \ldots, e_k). \) \hspace{1cm} (46)

\( V' = \nu'. \) \hspace{1cm} (47)

\( \sigma' = \sigma'_\nu. \) \hspace{1cm} (48)

By figures 3, 7, there exist normal type \( \nu'_u, \) and abnormal type \( E_\nu' \) such that \( T'_k = T_k' \parallel E_k' \) \( \sbullet \). Therefore, with (44) and (12), (9) and figure 7, by definition 13 and figure 8,

\( D \vdash T_k' \subseteq_c E_k, \) and \hspace{1cm} (49)

\( D \vdash T_k' \leq_w T_k' \) or \( T_k' = \bot \) and \( D \vdash T_k' \not\vdash \triangleq \) \hspace{1cm} (50)

1.1.2.1.1 case.

\( D \vdash T_k' \leq_w T_k. \) \hspace{1cm} (51)

With (50), (50), transitivity of widening we deduce that

\( D \vdash T_k' \leq_w T_k. \) \hspace{1cm} (51)

Applying lemma 12 and lemma 10 we obtain that

\( D, \nu', \sigma' \not\vdash t' : RetT(methH) \parallel (\cup E_\nu')_i \cup \cup E_\nu' \cup (\cup E_\nu')_i \cup \text{ExcT}(methH). \) \hspace{1cm} (53)

Define

\( T' = RetT(methH) \parallel (\cup E_\nu')_i \cup \cup E_\nu' \cup (\cup E_\nu')_i \cup \text{ExcT}(methH). \) \hspace{1cm} (54)

With (11), (54), (49), figure 8, definition 6 we have proven that \( D \vdash T' \leq_w T, \)

and with definition 13 that \( P \vdash T' \leq_w T. \)

Trivially, with (42), (43), (46), (47), (48), (53), definition 23, (54), by definition 24, \( P, \nu' \vdash t', \sigma' \leq T'. \)

59
1.1.2.1.2 case. $T^n_n = \bot$. The proof is similar to that of case 1.1.1.2 where
\[ \text{cont} \sqsubseteq \exists = e_0, \ldots, e_{n-1}, \mathcal{D} = \exists_e (e_1, \ldots, e_{k-1}, \mathcal{D} \sqsubseteq \exists_e, e_k, \ldots, e_n) \].

1.1.2.2 case. $e_0, \sigma', \sigma''$, and $P, \nu \vdash \sigma'' \sigma'$, and $T_k = \text{void}$ $\vdash \text{ET}_k$.

This case is not possible because $T_k$ is a component of an argument type: definition in figure 3 requires the argument type to be a variable type, therefore, $T_k \neq \text{void}$. $k \in \{1, \ldots, n\}$.

1.1.3 case.

\[ e_i \text{ is a value for } i \in \{0, \ldots, n\}, n \geq 0. \] (55)

As (14), we have the following two cases: 1.1.3.1 and 1.1.3.2.

1.1.3.1 case. $e_0 = \text{null}$. Applying the second rewriting rule from figure 26 we obtain:

\[ t, \sigma \vdash \text{throw new NullPE}(), \sigma' \]
Define the term $t'$, the state $\sigma'$, and the environment $\nu'$ as follows:

\[ t' = \text{throw new NullPE}(), \]
\[ \nu' = \nu, \]
\[ \sigma' = \sigma. \] (58)

As (1), $D$ includes $D_n$. From $D_n$ and applying subclass relationship and its transitivity, which are defined in figure 6, we deduce:

\[ D \vdash \text{nullPE} \sqsubseteq \text{nullPE}, \] (59)
\[ D \vdash \text{nullPE} \sqsubseteq \text{RTE}, \] (60)
\[ D \vdash \text{nullPE} \sqsubseteq \text{Throw} \] (61)

Apply the type rules for $\text{new} \ C()$ and $\text{throw} \ e$ with (56), (57), (58), (59), (61):

\[ D, \nu', \sigma' \vdash t' : \bot || \{\text{nullPE}\}. \] (62)

Define

\[ T' = \bot || \{\text{nullPE}\}. \] (63)

As (60) then with (11), by definition 6, we conclude:

\[ D \vdash \{\text{nullPE}\} \subseteq_e (\cup_{\text{ET}_i} \sqcap_{i=0}^n \cup \text{Ext}(\text{Method}) \]. (64)

Finally, with (11), (63), (64), by figure 8, we obtain that $D \vdash T' \leq \nu T$, and with definition 13 that $P \vdash T' \leq \nu T$.

With (57), (58), (2), (56), (62), definition 23, (63), by definition 24, $P, \nu' \vdash t', \sigma' \leq T'$.

1.1.3.2 case. $e_0 = \iota$. With (7) and the type rule for address in figure 28 we have that

\[ \sigma(\iota) = \ll(\ldots, \gg_{\mathbf{e}} \gg). \] (65)

With (9) and lemma 3, 2nd assertion, we obtain that

\[ \exists g^\mathbf{e} \text{ with } D \vdash g^\mathbf{e} \leq \nu g^\mathbf{e} \text{ and Method} \in \text{Method}(D, T_{0|^\mathbf{g}}, m). \] (66)

Therefore, by definition 5,

\[ \text{Method} \in \text{Method}(D, T_{0|^\mathbf{g}}, m). \] (67)

60
With (9) we have that

\[ \forall y T(\text{methH}) = T_1 \times \ldots \times T_n. \]  

(68)

Let

\[ RetT(\text{methH}) = T, \]  

(69)

\[ ExcT(\text{methH}) = \{ E_1, \ldots, E_q \}. \]  

(70)

Therefore, with (66), (67), (68), (69), (70), by lemma 6 and lemma 7, we obtain that \( \exists C', \text{meth}, T', E_T \) such that:

\[ p \vdash T_0 \subseteq C', \]  

(71)

\[ \text{meth} \in Meths(P, T_0, \mu) \text{ and meth} \in MDef(P, C', \mu), \]  

(72)

\[ \text{meth} = T = (T_1, p_1, \ldots, T_n, p_n) \text{ throws } E_1, \ldots, E_q \{ \text{Body} \}, \]  

(73)

\[ p, C' \text{ this; } T_1 p_1: \ldots; T_n p_n \text{ } \text{ throws } \text{Body} ; T' || E_T', \]  

(74)

\[ p \vdash T' || E_T' \leq_w T || \{ E_1, \ldots, E_q \}. \]  

(75)

With (55) and the rules for literals, identifiers, and addresses in figure 29, \( E_T \) = \emptyset for \( i \in \{ 0\ldots n \} \) in (11). Therefore:

\[ T = RetT(\text{methH}) || ExcT(\text{methH}) = T || \{ E_1, \ldots, E_q \}. \]  

(76)

With (72), (68), by definition 14, we obtain:

\[ \text{FirstFit}(P, \mu, T_0, T_1 \times \ldots \times T_n, \{ \text{meth} \}). \]  

(77)

Choose

\[ z_i, i \in \{ 0\ldots n \} \]  

(78)

With (5), (55), (65), (77), (73), (78), we can apply the third rewriting rule from figure 26 and obtain:

\[ t, \sigma(\text{Body})[z_0/\text{this}, z_1/p_1, \ldots, z_n/p_n], \sigma[z_0 \rightarrow e_0] \ldots [z_n \rightarrow e_n] \]  

(79)

Define the environment \( V' \), the state \( \sigma' \), and the term \( t' \) as follows:

\[ V' = V \cup C': z_0; T_1 z_1; \ldots; T_n z_n; \]  

(80)

\[ \sigma' = \sigma[z_0 \rightarrow e_0] \ldots [z_n \rightarrow e_n] \]  

(81)

\[ t' = \text{Body}[z_0/\text{this}, z_1/p_1, \ldots, z_n/p_n] \]  

(82)

By renaming \( \text{this} \) to \( z_0, p_i \) to \( z_i \), for \( i \in \{ 1\ldots n \} \) and (74), lemma 7, (80), (81), we obtain the following type for \( t' \):

\[ D, V', \sigma' \vdash t' : T' || E_T'. \]  

(83)

Define

\[ T' = T' || E_T' \]  

(84)

Then, by (75), (76) we obtain that \( p \vdash T' \leq_w T \).

As (78) and (2), then \( z_1 \) are also new in \( V \). With (81), (7), (8), (80), (71), (65) and widening definition in figure 8 for \( T_0 \), by definition 24, we obtain that \( p, V' \vdash \sigma' \emptyset \) and, therefore, \( p, V' \vdash t', \sigma' \leq T' \).
1.2 case. The last step in the normal typing of $t$ was the first assignment rule from figure 28 where $v$ is not an array access:

$$v \neq e'[e''] \text{ for any } e', e''.$$  \hspace{1cm} (85)

Therefore, $t$ has the form

$$t = v = e.$$  \hspace{1cm} (86)

So, the last step in the abnormal typing of $t$ is the assignment rule from figure 29. Hence, from figures 28, 29 we obtain:

$$D, V, \sigma \vdash v : T'' \parallel ET''.$$  \hspace{1cm} (87)
$$D, V, \sigma \vdash e : T' \parallel ET'.$$  \hspace{1cm} (88)
$$D \vdash T' \leq w T''.$$  \hspace{1cm} (89)

Thus, we apply the assignment type rule for $t$ and obtain:

$$D, V, \sigma \vdash t : \text{void} \parallel ET'' \cup ET'.$$  \hspace{1cm} (90)

Therefore,

$$T = \text{void} \parallel ET'' \cup ET'.$$  \hspace{1cm} (91)

Because of (89) and widening definition in figure 8,

$$T'' \neq \bot,$$  \hspace{1cm} (92)
$$T' \neq \bot.$$  \hspace{1cm} (93)

Hence, with (87) and (88), by (92) and (93), correspondingly,

$$v \neq \text{throw } \iota \text{ for any address } \iota,$$  \hspace{1cm} (94)
$$e \neq \text{throw } \iota \text{ for any address } \iota.$$  \hspace{1cm} (95)

1.2.1 case. $v$ is not l-ground. Therefore and because of (94), we apply the induction hypothesis, and with similar argumentation as in case 1.1.2.2 (namely, that $T$ is a variable type) we obtain only that $\exists \sigma''$, $\nu'$, $\nu''$, $T''$ such that:

$$v, \sigma'' \vdash \nu' \vdash \nu'' \vdash \sigma,'' \vdash T'',$$  \hspace{1cm} (96)
$$\nu'' \vdash \nu,$$  \hspace{1cm} (97)
$$P, \nu'' \vdash v'' \vdash \sigma'' \vdash T'' \text{ (i.e. } P, \nu'' \vdash \nu'' \vdash T''; \text{ and } P, \nu'' \vdash \sigma'' \vdash T''),$$  \hspace{1cm} (98)
$$P \vdash T'' \leq w T'' \parallel ET''.$$  \hspace{1cm} (99)

With (96), we apply the variable rewriting rule from figure 25:

$$t, \sigma'' \vdash v'' = e, \sigma''.$$  \hspace{1cm} (100)

Define the term $t'$, the state $\sigma'$, and the environment $\nu'$ as follows:

$$t' = \nu' = e,$$  \hspace{1cm} (100)
$$\nu' = \nu'',$$  \hspace{1cm} (101)
$$\sigma' = \sigma''.$$  \hspace{1cm} (102)

By figures 3, 7, there exist normal type $T''$, and abnormal type $ET''$ such that

$$T'' = T'' \parallel ET''.$$  \hspace{1cm} (103)

Therefore, with (99), (85), (87), (89), figure 7, by definition 13 and figure 8:

$$D \vdash ET'' \subseteq \nu \cdot ET'',$$  \hspace{1cm} (104)
$$D \vdash T'' \leq w T'' \text{ or } T'' = \bot.$$  

62
1.2.1.1 case.

\[ D \vdash T'' \leq_w T'''. \]  \hspace{1cm} (105)

With (85), (105), by induction hypothesis, \( T'' = T''' \). Because of (85), (96) and as a variable that is not a field access does not rewrite to a field access,

\[ \nu'' \neq e' [e'']' \]  \hspace{1cm} (106)

With (106), (98), (101), (102), (88), lemma 12 and lemma 10, (105), (100) we apply the first assignment type rule for \( t' \) and obtain:

\[ D, V', \sigma' \vdash t' : \text{void} \parallel E T'' \cup ET' \]  \hspace{1cm} (107)

Defining

\[ T' = \text{void} \parallel E T'' \cup ET', \]  \hspace{1cm} (108)

with (91), (104), figure 8, and definition 6 we have proven that \( D \vdash T' \leq_w T \), and with definition 13 that \( P \vdash T' \leq_w T \). Trivially, with (97), (98), (100), (100), (102), (107), definition 23, (108), by definition 24, \( P, V', T', \sigma' \leq T' \).

1.2.1.2 case. \( T'' = \perp \). The proof is similar to that of case 1.1.1.1 where

\[ \operatorname{cont} \sqsubseteq \Box = \square \cdot \Box = e. \]  \hspace{1cm} (109)

1.2.2 case.

\[ v \] is l-ground.

1.2.2.1 case. \( e \) is not ground. Therefore and because of (95), we apply the induction hypothesis, and with similar argumentation as in case 1.1.2.2 (namely, that \( T \) is a variable type and (89)) we obtain only that \( \exists \sigma'', v'', e'', T'' \) such that:

\[ e, \sigma'' \vdash v'' = e'', \]  \hspace{1cm} (110)

\[ v'' \leq v, \]  \hspace{1cm} (111)

\[ P, V'' \vdash t'' \leq T'' \] \( \text{i.e., } P, V, \sigma'' \vdash e'' : T'' \); and \( P, V'' \vdash \sigma'' \odot \), (112)

\[ P \vdash T'' \leq_w T' \parallel ET', \]  \hspace{1cm} (113)

With (110), we apply the expression rewriting rule from figure 25:

\[ t, \sigma'' \vdash v = e'', \sigma'' \]  \hspace{1cm} (114)

Define the term \( t' \), the state \( \sigma' \), and the environment \( V' \) as follows:

\[ t' = v = e'', \]  \hspace{1cm} (115)

\[ V' = v''. \]  \hspace{1cm} (116)

By figures 3, 7, there exist normal type \( T'' \), and abnormal type \( ET'' \) such that

\[ T'' = T' \parallel ET''. \]  \hspace{1cm} (117)

Therefore, with (113), (85), (88), (89), figure 7, by definition 13 and figure 8:

\[ D \vdash ET'' \leq_w ET', \]  \hspace{1cm} (118)

\[ D \vdash T'' \leq_w T' \text{ or } T'' = \perp. \]
1.2.2.1.1 case.

\[ D \vdash T'' \leq_w T'. \] (119)

With (89), (119) and transitivity of widening

\[ D \vdash T'' \leq_w T''. \] (120)

With (85), (112), (115), (116), (87), lemma 12 and lemma 10, (120), (114) we apply the first assignment type rule for \( t' \) and obtain:

\[ D, V', \sigma' \vdash t' : \text{void} \parallel ET'' \cup ET''. \] (121)

Defining

\[ T' = \text{void} \parallel ET'' \cup ET''. \] (122)

with (91), (118), figure 8, and definition 6 we have proven that \( D \vdash T' \leq_w T \), and with definition 13 that \( P \vdash T' \leq_w T \). Trivially, with (111), (112), (114), (115), (116), (121), definition 23, (122), by definition 24, \( P, V' \vdash t', \sigma' \not< T' \).

1.2.2.1.2 case. \( T^d = \perp \). The proof is similar to that of case 1.1.1.1.2 where

\[ \text{cont} \vdash \perp = v = \square \vdash \square. \] (123)

1.2.2.2 case. \( e \) is ground. Because of (93), \( e \) can not have the form \( \text{throw } i \), therefore \( e \) is a value.

As (85), (109) then, by definition 19, we have 3 cases: 1.2.2.2.1, 1.2.2.2.2, 1.2.2.2.3.

1.2.2.2.1 case. \( v \) is identifier. Apply the rewriting rule with variable as identifier from figure 25 and obtain that

\[ t, \sigma v = \sigma [v \rightarrow e]. \]

Defining \( \sigma' = \sigma [v \rightarrow e] \), with (87), (88), (89), (2) we obtain \( P, V \vdash \sigma' \Diamond \). With (90) the case is proven.

1.2.2.2.2 case. \( v = \iota.C[f] \). The proof is as in the previous case, where

\[ t, \sigma v = \sigma [v, f, C \rightarrow \text{val}], \]

\[ \sigma' = \sigma [v, f, C \rightarrow \text{val}]. \]

1.2.2.2.3 case. \( v = \text{null}.[C]f \). Therefore, \( t \) has the form \( \text{null}.[C]f = e \). With (123) we apply the assignment rewrite rule with field access to \( \text{null} \) from figure 25 and proof as in case 1.1.3.1 where with (91)

\[ D \vdash \{ \text{null} \}_E \subseteq \iota, \text{ET} \cup ET'. \]

1.3 case. The last step in the normal typing of \( t \) was the second assignment rule from figure 28 where the left-hand side variable is an access to an array of primitive types. Therefore, \( t \) has the form

\[ t = e_1[e_2] = e_3. \] (124)

So, the last step in the abnormal typing of \( t \) is the assignment rule from figure 29. Hence, from figures 28, 29 with account of type rules for array access we obtain:

\[ \vdash T_1[] \Diamond \text{prim Av} \] (125)

\[ D, V, \sigma \vdash e_1 : T_1[] \parallel ET_1 \] (126)

\[ D, V, \sigma \vdash e_2 : \text{int} \parallel ET_2 \] (127)

\[ D, V, \sigma \vdash e_1[e_2] : T_1 \parallel ET_1 \cup ET_2 \] (128)

\[ D, V, \sigma \vdash e_3 : T_3 \parallel ET_3 \] (129)

\[ D \vdash T_3 \leq_w T_1. \] (130)
Applying the second assignment type rule for \( t \) we obtain:

\[
D, \nu, \sigma \not\in \tau : \texttt{void} \parallel ET_1 \cup ET_2 \cup ET_3.
\] (131)

Therefore,

\[
T = \texttt{void} \parallel ET_1 \cup ET_2 \cup ET_3.
\] (132)

Because of (130), definition of array type in figure 1 and widening definition in figure 8.

\[
T_1 \neq \bot, \quad T_3 \neq \bot.
\] (133)

Hence, with (126), (127), (129), by (133) and (134), correspondingly for the \( e_1 \) and \( e_3 \)

\[
e_i \neq \text{throw } i \text{ for any address } i \text{ where } i \in \{1...3\}.
\] (135)

1.3.1 case. \( e_1[e_2] \) is not I-ground.

1.3.1.1 case. \( e_1 \) is not ground. Therefore we apply the induction hypothesis, and with similar argumentation as in case 1.1.2.2 (namely, that \( T_1 \) is a primitive type) we obtain only that \( \exists \sigma''', \nu'', e''', T''_1 \) such that:

\[
e_1, \sigma''', e''', T''_1.
\] (136)

\[
\nu'' \subseteq \nu.
\] (137)

\[
P, \nu'' \vdash e''', \sigma''' \sqsubseteq T''_1 (i.e. P, \nu'', \sigma''' \not\in \nu'' : T''_1); \text{ and } P, \nu'' \vdash \sigma''' \diamond.
\] (138)

\[
P \vdash T''_1 \leq_w T_1 \parallel ET_1.
\] (139)

With (136), we apply the array expression rewriting rule from figure 23:

\[
t, \sigma''', e''', T''_1 \equiv e_1, \sigma''''.
\] (140)

Define the term \( t' \), the state \( \sigma' \), and the environment \( \nu' \) as follows:

\[
t' = e''', \quad \nu' = \nu'', \quad \sigma' = \sigma''''.
\] (141)

(142)

By figures 3, 7, there exist normal type \( T''_1 \), and abnormal type \( ET''_1 \) such that

\[
T''_1 = T''_1 \parallel ET''_1.
\] (143)

Therefore, with (139), (126) and figure 7, by definition 13 and figure 8:

\[
D \vdash ET''_1 \subseteq_c ET_1
\] (144)

\[
D \vdash T''_1 \leq_w T_1 \parallel \text{or } T''_1 = \bot.
\] (145)

1.3.1.1.1 case.

\[
D \vdash T''_1 \leq_w T_1 \parallel.
\] (146)

Therefore, with (125), by figure 8:

\[
T''_1 = T_1 \parallel
\] (147)

Consequently,

\[
T''_1 \diamond \text{prim.Ar}
\] (148)
With (138), (143), (147), (141), (142), (127), lemma 12 and lemma 10, applying the access type rule we obtain that

\[ D, V', \sigma' \not\in e'_1[e_2] : T_1 \rightarrow ET''_1 \cup ET_2, \]

and then with the latter and (129), lemma 12 and lemma 10, (130), (148) we apply the corresponding (second in this case) assignment type rule for \( t' \) and obtain:

\[ D, V', \sigma' \not\in t' : \text{void} \rightarrow ET''_1 \cup ET_2 \cup ET_3 \]

Defining

\[ T' = \text{void} \rightarrow ET''_1 \cup ET_2 \cup ET_3, \]

with (132), (144), figure 8, and definition 6 we have proven that \( D \vdash T' \leq_w T \), and with definition 13 that \( P \vdash T' \leq_w T \). Trivially, with (137), (138), (140), (141), (142), (149), definition 23, (150), by definition 24, \( P, V' \vdash t', \sigma' \leq T' \).

**1.3.1.1.2 case.**

\[ T''_1 = \perp. \]

Define a context \( \text{cont}' \sqsubseteq \square \) and \( t'' \) such that

\[ \text{cont}' \sqsubseteq \square = \square \rightarrow [e_2], \]

\[ t'' = \text{cont}' \sqsubseteq \square. \]

Applying the context type rule for \( t'' \) from figures 28, 29, and with (138), (143), (151), (141), (142), we obtain the following type for \( t'' \):

\[ D, V', \sigma' \not\in t'' : \perp \rightarrow ET''_1. \]

Then defining

\[ \text{cont} \sqsubseteq \square = \square \rightarrow e_3, \]

\[ t' = \text{cont} \sqsubseteq t'' \]

we prove the case similarly to case 1.1.1.1.2.

**1.3.1.2 case.**

\[ e_1 \text{ is ground}, \]

\[ e_2 \text{ is not ground}. \]

Because of (135) with (153), \( e_1 \) is a value. With (126), by definition 19,

\[ e_1 = \text{null} \ 	ext{or} \ e_1 \text{ is an address}. \]

As (154) and (135) with \( i = 2 \), we apply the induction hypothesis, and because of (127), we obtain only that \( \exists \sigma'', V'', e''_2, T''_2 \) such that:

\[ e_2, \sigma'' \not\in e''_2, \sigma'', \]

\[ V'' \leq V, \]

\[ P, V'' \vdash e''_2, \sigma'' \ 	ext{and} \ T''_2 \ (i.e. P, V'' \not\in e''_2 : T''_2; \ 	ext{and} P, V'' \vdash \sigma'' \diamond), \]

\[ P \vdash T''_2 \leq_w \text{int} \rightarrow ET_2. \]

With (156) correspondingly to the forms of \( e_1 \) given in (155) we apply either the \( \text{null} \) or the address array expression rewriting rule from figure 23 and obtain:

\[ t, \sigma'' \not\in e''_1[e''_2] = e_3, \sigma''. \]
Define the term \( t' \), the state \( \sigma' \), and the environment \( \mathcal{V}' \) as follows:

\[
\begin{align*}
t' &= e_1[e_2'] = e_3, \\
\mathcal{V}' &= \mathcal{V}'', \\
\sigma' &= \sigma''.
\end{align*}
\]

By figures 3, 7, there exist normal type \( T_2'' \) and abnormal type \( ET_2'' \) such that

\[
T_2'' = T_2'' \parallel ET_2''.
\]

Therefore, with (159) and figure 7, by definition 13 and figure 8:

\[
\begin{align*}
D \vdash ET_2'' \subseteq e \ ET_2 \quad & (164) \\
D \vdash T_2'' \subseteq_w \text{int} & \text{or } T_2'' = \bot \quad \text{and } D \vdash \text{int} \Diamond \text{NotType}.
\end{align*}
\]

1.3.1.2.1 case.

\[
D \vdash T_2'' \subseteq_w \text{int}. \quad (165)
\]

Therefore, by figure 8

\[
T_2'' = \text{int}. \quad (166)
\]

With (158), (163), (166), (161), (162), (126), lemma 12 and lemma 10, applying the array access type rule we obtain that

\[
D, \mathcal{V}', \sigma' \not\vdash e_1[e_2'] : T_1 \parallel ET_1 \cup ET_2''.
\]

The rest of the proof as in the case 1.3.1.1.1 where

\[
D, \mathcal{V}', \sigma' \not\vdash t' : \text{void} \parallel ET_1 \cup ET_2'' \cup ET_3,
\]

\[
T' = \text{void} \parallel ET_1 \cup ET_2'' \cup ET_3
\]

1.3.1.2.2 case. \( T_2'' = \bot \). The proof – as in the case 1.3.1.1.2 where

\[
\text{cont}'[\bot] = e_2[\bot].
\]

1.3.2 case. \( e_1[e_2] \) is l-ground. This means that (153) and (155), and also

\[
e_2 \text{ is ground.} \quad (167)
\]

1.3.2.1 case. \( e_2 \) is not ground. The proof is similar to that of the case 1.2.2.1 with adjusting the application of the corresponding assignment type rule to \( t' \).

1.3.2.2 case. \( e_3 \) is ground. Therefore, with (135),

\[
e_3 \text{ is a value.} \quad (168)
\]

With (155) we obtain two cases: 1.3.2.2.1 and 1.3.2.2.2.

1.3.2.2.1 case. \( e_1 = \text{null} \). Therefore, \( t \) has the form \( \text{null}[e_2] = e_3 \). With (167), (168) we apply the assignment rewrite rule with \( \text{null} \) array from figure 25 and proof as case 1.1.3.1 where with (132) \( D \vdash \{\text{NullPE} \} \subseteq e \ ET_1 \cup ET_2 \cup ET_3 \).

1.3.2.2.2 case.

\[
e_1 \text{ is an address.} \quad (169)
\]

As (2), (126), (169) then by definition 24:

\[
\sigma(e_1) = [\text{val}_0, \ldots, \text{val}_{n-1}]^{\top},
\]

(170)
1.3.2.2.1 case.

\[ 0 > e_2 \text{ or } e_2 > n - 1. \quad (171) \]

With (168), (167), (170), (171), (124) we apply the "out of bound" type rule from figure 25 and obtain

\[ t, \sigma \vdash \text{throw new IndOutBndE}(), \sigma \]

The rest of the proof is similar to that of case 1.1.3.1 where

\[ t' = \text{throw new IndOutBndE}(). \]
\[ D \vdash \text{IndOutBndE} \subseteq \text{RTE} \]
\[ D, V', \sigma' \vdash t' : \bot \parallel \text{IndOutBndE} \]
\[ T' = \bot \parallel \text{IndOutBndE} \]
\[ D \vdash \text{IndOutBndE} \subseteq \bigcup \text{ET}_1 \cup \text{ET}_2 \cup \text{ET}_3 \]

1.3.2.2.2 case.

\[ 0 \leq e_2 \leq n - 1 \quad (172) \]

As (125), by definition 21,

\[ e_3 \text{ fits } T_1[i] \text{ in } P \text{ and } \sigma. \quad (173) \]

Therefore, with (168), (167), (170), (172), (173), (124) we apply the "fits" rewrite rule from figure 25 and obtain:

\[ t, \sigma \vdash e_1, e_2 \mapsto e_3 \]

Define

\[ \sigma' = \sigma[e_1, e_2 \mapsto e_3] \]

Because (130), (125), (129), \( e_3 \) is a primitive value, and

\[ P, \sigma' \vdash e_3 \ll_1 T_1 \quad (174) \]

Taking \( V' = V \) with (2), (170), (174), we obtain \( P, V' \vdash \sigma' \Diamond \). With (131) we have proven the case.

1.4 case. The last step in the normal typing of \( t \) was the third assignment rule from figure 28 where left-hand side variable is an access to an array of reference types. Therefore, \( t \) has the form

\[ t = e_1[e_2] = e_3. \quad (175) \]

So, the last step in the abnormal typing of \( t \) is the assignment rule from figure 29. Hence, from figures 28, 29 with account of type rules for array access we obtain:

\[ \vdash T_1[i] \ll_{\text{prim, Ar}} \quad (176) \]
\[ D, V, \sigma \vdash e_1 : T_1[i] \parallel ET_1 \quad (177) \]
\[ D, V, \sigma \vdash e_2 : \text{int} \parallel ET_2 \quad (178) \]
\[ D, V, \sigma \vdash e_1[e_2] : T_1 \ll ET_1 \cup ET_2 \quad (179) \]
\[ T_3 \neq \bot \quad (180) \]
\[ D, V, \sigma \vdash e_3 : T_3 \ll ET_3. \quad (181) \]

Applying the third assignment type rule for \( t \) we obtain (131) and, therefore, take (132). Because of (177), (178), (180) and (181), we conclude that

\[ e_1 \not= \text{throw } \iota \text{ for any address } \iota \text{ where } \iota \in \{1..3\}. \quad (182) \]
1.4.1 case. \( e_1[e_2] \) is not 1-ground.

1.4.1.1 case. \( e_1 \) is not ground. Therefore and because of (182) with \( i = 1 \), we apply the induction hypothesis, and with similar argumentation as in case 1.1.2.2 (namely, that \( T_1 \) is a reference type by (176)) and obtain only that \( \exists \sigma'', \psi'', e''_1, T''_1 \) such that (136), (137), (138), (139). With (136), we apply the array expression rewriting rule from figure 23 and obtain (140). The term \( \psi' \), the state \( \sigma' \), and the environment \( \psi' \) are defined the same way as in (140), (141), (142). Taking \( T''_1 \) as in (143) we come to (144), (145) and to the two subcases as follows.

1.4.1.1.1 case.

\[
D \vdash T''_1 \leq_w T_1].
\]

Therefore, by figure 8, we obtain that there exists \( T''_1 \) such that

\[
T''_1 = T_1], \quad \text{and} \quad D \vdash T''_1 \leq_w T_1].
\]

(183)

With (182), (184) we obtain

\[
\psi' T''_1 \circ \text{primAr}
\]

(185)

With (138), (143), (183), (141), (142), (178), lemma 12 and lemma 10, applying the array access type rule we obtain that

\[
D, \psi', \sigma' \not\in e''_1, e''_1 : T_1 \parallel ET''_1 \cup ET_2.
\]

and then with the latter and (181), (185), lemma 12 and lemma 10, (176) we apply the third assignment type rule to \( \psi' \) and obtain (149). Defining \( T' \) as in (150) the rest of the proof is as that of the proof in case 1.3.1.1.1.

1.4.1.1.2 case. \( T''_1 = \perp. \) The proof is as that for case 1.3.1.1.2.

1.4.1.2 case. \( e_1 \) is ground, \( e_2 \) is not ground. The proof is similar to that for case 1.3.1.2 with the corresponding rest of the proof as in case 1.4.1.1.1.

1.4.2 case. \( e_1[e_2] \) is 1-ground. This means that (153) with (155), and (167).

1.4.2.1 case. \( e_1 \) is not ground. Therefore we apply the induction hypothesis, and with similar argumentation as in case 1.1.2.2 (namely, that \( T_0 \) is a reference type) we obtain only that \( \exists \sigma'', \psi'', e''_1, T''_1 \) such that:

\[
e''_2, \sigma''.
\]

(186)

\[
\psi'' \leq \psi.
\]

(187)

\[
P, \psi'' \vdash T''_1 \quad (i.e. \ P, \psi'' \not\in e''_1 : T''_1, \quad \text{and} \quad P, \psi'' \vdash \sigma'' \circ \}.)(188)
\]

\[
P \vdash T''_1 \leq_w T_3 || ET_3.
\]

(189)

With (186), we apply the expression rewriting rule from figure 25:

\[
t, \sigma_2 \circ e_1 = e''_1, \sigma''.
\]

Define the term \( \psi' \), the state \( \sigma' \), and the environment \( \psi' \) as follows:

\[
\psi' = e_1[e_2] = e''_1.
\]

(190)

\[
\psi'' = \psi''.
\]

(191)

\[
\sigma' = \sigma''.
\]

(192)

By figures 3, 7, there exist normal type \( T''_3 \), and abnormal type \( ET''_3 \) such that

\[
T''_3 = T''_3 || ET''_3.
\]

(193)
Therefore, with (189) and figure 7, by definition 13 and figure 8:

\[ D \vdash E_{T_3} \subseteq T_3 \]
\[ D \vdash \tau'' \leq T_3 \text{ or } \tau'' = \bot. \]  \hspace{1cm} (194)

1.4.2.1.1 case.

\[ D \vdash T_3 \leq T_3 \]  \hspace{1cm} (195)

Therefore, by figure 8,

\[ T'' \neq \bot \]  \hspace{1cm} (196)

With (196), (188), (191), (192), (179), lemma 12 and lemma 10, (176), (190) we apply the third assignment type rule to \( \tau' \) and obtain that

\[ D, \tau', \sigma' \vdash \text{void} \| ET_1 \cup ET_2 \cup ET_3'' \]. \hspace{1cm} (197)

Defining

\[ T' = \text{void} \| ET_1 \cup ET_2 \cup ET_3'' \] \hspace{1cm} (198)

with (132), (194), figure 8, and definition 6 we have proven that \( D \vdash T' \leq T \), and with definition 13 that \( P \vdash T' \leq T \). Trivially, with (187), (188), (190), (191), (192), (197), definition 23, (198), by definition 24, \( P, \tau' \vdash \tau', \sigma' \subseteq T' \).

1.4.2.1.2 case. \( T_3'' \neq \bot \). The proof is as in case 1.2.2.1.2.

1.4.2.2 case. \( e_3 \) is ground. Therefore with (182) for \( i=3 \) we obtain that \( e_3 \) is a value. By definition 19, we have two subcases:

1.4.2.2.1 case. \( e_4 \) is null. The proof is as in case 1.3.2.2.1.

1.4.2.2.2 case. \( e_4 = t \) and

\[ \sigma(e_4) = [v_{a_1}, \ldots, v_{a_n}]^{\text{val}} \].

1.4.2.2.2.1 case. \( 0 \leq e_2 \leq n - 1 \). The proof is as in case 1.3.2.2.2.1.

1.4.2.2.2.2 case. \( 0 > e_2 \) or \( e_2 > n - 1 \). The proof is as in case 1.3.2.2.2.2.

1.4.2.2.2.2.1 case. \( e_3 = \text{null} \). With definition 21 and (176), the proof as in case 1.3.2.2.2.2.

1.4.2.2.2.2.2 case. \( e_3 \) is an address. Therefore, with (181), \( T_3 \) is either class or array type.

1.4.2.2.2.2.2.1 case.

\[ P \vdash T_3 \leq T_1 \] \hspace{1cm} (199)

If \( P \vdash T_3 \subseteq T_3 \) then

\[ \sigma(e_3) = \ll \ldots \gg_{T_3} \] \hspace{1cm} (200)

and with (199) \( P \vdash T_3 \subseteq T_1 \). Therefore, by definition 21,

\[ e_3 \text{ fits } T_1 [] \text{ in } P \text{ and } \sigma \], and \hspace{1cm} (201)

\[ P, \sigma \vdash e_3 <_{t} T_1 [] \] \hspace{1cm} (202)

If \( T_3 = T_4 [] \ldots [] \) then

\[ \sigma(e_3) = [\ldots T_3 = [\ldots T_4 [] \ldots []], \] \hspace{1cm} (203)

\[ e_3 \text{ fits } T_1 [] \text{ in } P \text{ and } \sigma \], and \hspace{1cm} (204)

\[ P, \sigma \vdash e_3 <_{t} T_1 [] \] \hspace{1cm} (205)
Therefore, with (168), (167), (200) or (203), (202) or (205) (172), (201) or (204), (175) we apply the "fits" rewrite rule from figure 25 and complete the proof as in case 1.4.2.2.2.2.

**1.4.2.2.2.2 case.** Let \( P \not\vdash T_3 \leq_{w} T_1 \). Therefore, with definition 21,

\[
e_a \text{ does not fit } T_1[] \text{ in } P \text{ and } \sigma. \tag{206}
\]

Therefore, with (168), (167), (200) or (203), (172), (206), (175) we apply the "does not fit" rewrite rule from figure 25 and obtain:

\[
t, \sigma \not\vdash \text{throw new } \text{ArrStoreE}, \sigma
\]

The rest of the proof is similar to that of case 1.1.3.1 where

\[
t' = \text{throw new } \text{ArrStoreE},
\]

\[
 D \vdash \text{ArrStoreE} \subseteq \text{RTE}
\]

\[
 D, \nu', \sigma' \not\vdash t' : \bot \cup \{\text{ArrStoreE}\}
\]

\[
 T' = \bot \cup \{\text{ArrStoreE}\}
\]

\[
 D \vdash \{\text{ArrStoreE}\} \subseteq _e ET_1 \cup ET_2 \cup ET_3
\]

**1.4.2.2.2.3 case.** \( e_a \) is an integer or character, or boolean. This implies that \( P \not\vdash T_3 \leq_{w} T_1 \) and the rest as in case 1.4.2.2.2.2

**2 case.**

\[
 T = \bot \tag{207}
\]

By lemma 9 we have two cases: 2.1 and 2.2.

**2.1 case.** \( \exists E', ET' \) such that

\[
 D \vdash E' \subseteq \text{Throwable}, \tag{208}
\]

\[
 D \vdash ET' \odot \text{ArrType}, \tag{209}
\]

\[
 t = \text{throw } e.
\]

\[
 D, \nu, \sigma \not\vdash e : E' \cup ET', \tag{210}
\]

\[
 ET = \{E'\} \cup ET'. \tag{211}
\]

Because of \( (208) \) and figure 8, \( E' \neq \bot \), and because of \( (210) \)

\[
 e \neq \text{throw } e'. \tag{212}
\]

**2.1.1 case.** \( e \) is ground. As, by \( (210), (208) \), \( e \) has a class type, therefore, by definition 19, we have two subcases: 2.1.1.1 and 2.1.1.2.

**2.1.1.1 case.** \( e = \text{null} \). By the type rule for \text{throw null} from figure 29, \( ET = \{\text{NullPE}\} \). Apply the \text{throw null} rewrite rule from figure 27:

\[
t, \sigma \not\vdash \text{throw new } \text{NullPE}, \sigma
\]

The rest of the proof as in case 1.1.3.1 where

\[
 D \vdash \{\text{NullPE}\} \subseteq _e \{\text{NullPE}\}.
\]
2.1.1.2 case. \( e = \iota \).

This case is not applicable since \( t \) would have the form \( \text{throw} \ \iota \), and so would be ground.

2.1.2 case. \( e \) is not ground. As (212), we apply the induction hypothesis, and with similar argumentation as in cases 1.1.1.2 or 1.1.2.2 (namely, (208), (210) for this case) we obtain only that \( \exists \sigma'', \nu'' : e'' \mapsto T'' \) such that:

\[
\begin{align*}
\sigma, \sigma'' & \in e''', \sigma''', \quad(213) \\
\nu'' & \subseteq \nu, \quad(214) \\
P, \nu'' \vdash e'', \sigma'' \mapsto T'' \quad (i.e. \ P, \nu'', \sigma'' \not\vdash e'' : T''; \ and \ P, \nu'' \vdash \sigma'' \diamondsuit), \quad(215) \\
P \vdash T'' \leq_w E' || ET'. \quad(216)
\end{align*}
\]

With (213), we apply the \( \text{throw} \ e \) rewrite rule from figure 27 and obtain:

\[
t, \sigma \mapsto \text{throw} \ e'', \sigma'' \quad(217)
\]

Define

\[
\begin{align*}
t' &= \text{throw} \ e'' \quad(218) \\
\nu' &= \nu'', \quad(219) \\
\sigma' &= \sigma'' \quad(220)
\end{align*}
\]

By figures 3, 7, there exist normal type \( T''' \), and abnormal type \( ET'' \) such that \( T'' = T'' || ET'' \). Therefore, with (216), (210), by definition 13 and figure 8:

\[
\begin{align*}
D \vdash ET'' & \subseteq_e ET', \quad(221) \\
D \vdash T'' & \leq_w E' \quad \text{or} \quad T'' = \perp. \quad(222)
\end{align*}
\]

2.1.2.1 case.

\[
D \vdash T'' \leq_w E. \quad(223)
\]

Therefore, with (208) we also have that

\[
D \vdash T'' \subseteq \text{Throwable}. \quad(224)
\]

Now applying the type rules for \( \text{throw} \ e'' \) from figures 28, 29 with (219), (220), (224), (225), we obtain

\[
D, \nu', \sigma' \not\vdash t' : \perp || \{ T'' \} \cup ET'', \quad(225)
\]

Because of (211), (221), (223),

\[
D \vdash \{ T'' \} \cup ET'' \subseteq_e ET \quad(226)
\]

Therefore, taking

\[
T' = \perp || \{ T'' \} \cup ET'', \quad(227)
\]

we prove, by figure 8, that \( D \vdash T' \leq_w T \), and with definition 13 that \( P \vdash T' \leq_w T \).

Finally, with (214), (215), (218), (219), (220), (225), definition 23, (227), by definition 24, \( P, \nu' \vdash t', \sigma' \subseteq T' \).

2.1.2.2 case. \( T' = \perp \). This means that \( e'' \) is source of another exception. The proof is similar to that of case 1.1.1.1.2 where

\[
\text{cont} \cdot \square = \text{throw} \cdot \square. \quad(228)
\]

Particularly, with (221) we prove that \( P \vdash T' \leq_w T \).

72
2.2 case. ∃ context cont □ : □ and term \( t'' \):

\[
t = \text{cont □ } t'' □ \quad \text{and} \quad \text{D, } \forall, \sigma \vDash t'' : T
\]

(228)

(229)

2.2.1 case. \( t'' \) is a value. The case is not applicable because of (229), (3), (207).

2.2.2 case.

\[
t'' = \text{throw } \iota.
\]

(230)

Therefore, with (228)

\[
t = \text{cont □ } \text{throw } \iota □
\]

Applying the rewrite rule for \( \text{cont □ } \text{throw } \iota □ \) from figure 27 we obtain:

\[
t, \sigma \vDash t''', \sigma
\]

Define

\[
t' = t''.
\]

(231)

Therefore, with (229) they have the same type, as the type of \( t \) (c.f. (3), (207)). Define

\[
T' = T.
\]

(232)

Therefore, with definition 13, by figure 8, \( P \vdash T' \leq_v T \). Trivially, because in this case \( \forall' = \forall \) and \( \sigma' = \sigma \), therefore, with (2), (231), (229), definition 23, (232), by definition 24, \( P, \forall' \vdash t', \sigma' \leq T' \).

2.2.3 case. \( t'' \) is not ground. Therefore, the induction hypothesis is applicable; with similar argumentation as in cases 1.1.1.2, 1.1.2.2, or 2.1.2 (namely, (229), (3), (207) for this case) we obtain only that \( \exists \sigma', \forall', t'''', T''' \) such that:

\[
t'', \sigma \vDash t'''', \sigma',
\]

(233)

\[
\forall' \leq \forall,
\]

(234)

\[
P, \forall' \vdash t'''', \sigma' \leq T''' \text{ (i.e. } P, \forall', \sigma' \vDash t'''', T''' ; \text{ and } P, \forall' \vdash \sigma' \triangleleft),
\]

(235)

\[
P \vdash T''' \leq_v T.
\]

(236)

As (236), (207), (3) then with definition 13, by figures 8, 3, 7, there exists abnormal type ET' such that

\[
T''' = \bot \parallel ET', \quad \text{and}
\]

\[
P \vdash ET' \subseteq_e ET.
\]

(237)

(238)

Now we continue with case analysis over the structure of \( \text{cont □ } \cdot \cdot □ \) according to figure 21.

2.2.3.1 case. \( \text{cont □ } \cdot \cdot □ \) is a field access context:

\[
\text{cont □ } \cdot \cdot □ = \text{□ } \cdot \cdot □ , [C] f.
\]

(239)

Therefore,

\[
t = t''', [C] f
\]

(240)

With (233) we apply the rewrite rule for field access and obtain that

\[
t, \sigma \vDash t''', [C] f, \sigma'
\]

(241)
Define
\[ t' = t''[C][f] \] (242)

\( t' \) is also a field access context and has the same form cont \( [\cdot : \Box] \) as in (239), and
\[ t' = \text{cont} \ [\cdot : \Box][f] \]

Therefore, with (234), (235), (237), definition 23, and lemmas 12, 10 we apply the context type rules from figures 28, 29 and obtain:
\[ P, V', \sigma' \vdash t' : \perp \downarrow ET' \] (243)

Define
\[ T' = \perp \downarrow ET' \] (244)

With (238) and definition 13, by figure 8. \( P \vdash T' \leq w \ T \).

Finally, with (234), (235), (242), (243), (244), by definition 24, \( P, V' \vdash t', \sigma' \vdash T' \).

2.2.3.2 case. cont \( [\cdot : \Box] = [\cdot : \Box][e] \).
The proof is similar to that in case 2.2.3.1.

2.2.3.3 case. cont \( [\cdot : \Box] = e[\cdot : \Box] \).
The proof is similar to that in case 2.2.3.1.

2.2.3.4 case. cont \( [\cdot : \Box] = \text{new } T [e_1 \ldots [\cdot : \Box][k \ldots e_n] \).
The proof is similar to that in case 2.2.3.1.

2.2.3.5 case. cont \( [\cdot : \Box] = e[\cdot : \Box][\text{AT}] \).
The proof is similar to that in case 2.2.3.1.

2.2.3.6 case. cont \( [\cdot : \Box] = e[\cdot : \Box][\text{AT}]m(e') \).
The proof is similar to that in case 2.2.3.1.

where \( n \geq 1, 1 \leq k \leq n \).
The proof is similar to that in case 2.2.3.1.

2.2.3.7 case. cont \( [\cdot : \Box] = e \).
The proof is similar to that in case 2.2.3.1.

2.2.3.8 case. cont \( [\cdot : \Box] = \text{if } [\cdot : \Box] \text{ then stats else stats} \).
The proof is similar to that in case 2.2.3.1.

2.2.3.9 case. cont \( [\cdot : \Box] = [\cdot : \Box] ; \text{return } e \).
The proof is similar to that in case 2.2.3.1.

2.2.3.10 case. cont \( [\cdot : \Box] = [\cdot : \Box] ; \text{stat} \).
The proof is similar to that in case 2.2.3.1.

2.2.3.11 case. cont \( [\cdot : \Box] = \text{return } [\cdot : \Box] \).
The proof is similar to that in case 2.2.3.1.

2.2.3.12 case. cont \( [\cdot : \Box] = \text{throw } [\cdot : \Box] \).
The proof is similar to that in case 2.2.3.1.

\( \square \)

The method call from the philosophers example aPhil\([\text{Quest}]\) think(a\text{Quest}) is type checked in the environment \( \psi_{ph} \) from section 13.1. When it gets evaluated (c.f. the rewrites in section 11), then after the third rewrite step, the “environment extension” required by the subject reduction theorem is \( \psi'_{ph} = \psi_{ph} \text{ FrPhil } \text{z0: Quest } \text{z1} \). The states \( \sigma_1, \sigma_2 \) conform to \( \psi'_{ph} \) and \( \text{B}_{ph} \), and \( \psi'_{ph} \leq \psi_{ph} \).

Finally, the soundness theorem states that execution of a well-formed Java\( ^b \) program will produce a value of the expected normal type, or it will terminate in new state, or it will not terminate, or it will terminate with a thrown exception. This exception will either be an unchecked exception, or it will belong to a subclass of one of the classes included in the abnormal type of the term. In all cases the final state will conform to the Java\( ^b \) program.

74
Theorem 2 Soundness Take any Java\textsuperscript{a} term \(t\) and any Java\textsuperscript{b} program \(P\) with \(\vdash P \diamond\), environment \(V\) with \(P \vdash V \diamond\), type \(T = T \parallel ET\), and state \(\sigma\) with \(P, V \vdash t, \sigma < T\). Then there exists a unique Java\textsuperscript{a} term \(t'\), type \(T'\) and state \(\sigma'\), such that either

- \(T \neq \text{void}, t.\sigma_{\gamma}^* \cdot t', \sigma', \ t'\) is ground, \\
  \(\exists T': P, V \vdash t', \sigma' \prec T'\) and \(P \vdash T' \leq_w T\), or
- \(T = \text{void}, t.\sigma_{\gamma}^* \cdot \sigma'\) and \(P, V \vdash \sigma' \Diamond\), or
- \(t.\sigma_{\gamma}^* \text{ does not terminate}\), or

- \(\tau.\sigma_{\gamma}^* \text{ throw } i, \sigma', P, V \vdash \sigma' \Diamond\), \(\sigma'(i) = \ll \ldots \rr\) and \(P \vdash \{E\} \subseteq_e ET \cup RTE \cup Error\).

An implication of the soundness theorem is that if a term has type \(\perp \parallel ET\) then if its execution terminates, it will produce an exception of a class which has a superclass in \(ET\), or which is an unchecked exception.

For the worries example, the environment \(V_{\text{wr}}\), introduced in 3.5, the soundness theorem and the typing

\[ B_3B_3V_{\text{wr}} \vdash \text{pete}_{\text{wr}} \cdot \text{[[act]}}()\cdot \sigma_{10} \prec \text{void} \parallel \{\text{Worry, Illness}\} \]

guarantee that execution of the term \(\text{pete}_{\text{wr}} \cdot \text{[[act}()\cdot \text{in state} \sigma_{10}\text{ will either terminate successfully, or throw an exception of class Worry or Illness (or a subclass), or throw an unchecked exception such as NullPointerException or OutOfMemoryError. As we outlined in section 11.2, if we disregard the possibility of unchecked exceptions, there are three possible outcomes, namely } \sigma_{11}, \sigma_{14}, \text{ and throw } \tau_{14}, \tau_{13}\text{ with } \sigma_{13}(\tau_{13}) = \ll \ldots \rr_{\text{Worry}}. \text{ For these it holds that:}

- \(B_3B_3V_{\text{wr}} \vdash \sigma_{11} \Diamond;\)
- \(B_3B_3V_{\text{wr}} \vdash \sigma_{14} \Diamond;\)
- \(B_3B_3V_{\text{wr}} \vdash \sigma_{13} \Diamond, \text{ and } B_3B_3V_{\text{wr}} \vdash \{\text{Worry}\} \subseteq_e \{\text{Worry, Illness}\}.\)

14 Java as a fragment system

In \cite{9}, Sophia Drossopoulou et al. introduced fragment systems, which reason about separate compilation and linking at a high, language independent level. They used fragment systems to explore possible meanings of binary compatibility and their properties.

In this section we demonstrate how Java\textsuperscript{a} and Java\textsuperscript{b} can be understood as a fragment system.

14.1 Fragment systems

Fragment systems, as defined in \cite{9}, aim to model separate compilation and linking. Fragments are the basic units participating in compilation and linking. We distinguish L\textsuperscript{b} fragments and L\textsuperscript{a} fragments, where:

- \(L^a\) stands for the source language,
- \(L^b\) stands for the binary language containing all necessary information for execution and for compilation of importing fragments.

We repeat the definitions from \cite{9}.
Definition 25 A tuple \((L^a, L^b, \epsilon, \text{Ids}^a, C^a, \perp^a, T^a, \preceq^a, \nabla^a, \odot^a)\) is a fragment system, iff

- \(L^a, L^b, \text{Ids}^a\) are sets, and \(L^a \cap L^b = \{\epsilon\}\),
- \(C^a\) is a mapping, \(C^a : L^a \times L^b \rightarrow L^a\),
- \(\perp^a\) is a commutative, associative operator; \(\perp^a : L^a \times L^b \rightarrow L^a \cup L^b \times L^b \rightarrow L^a\), with \(\epsilon\) as the identity element,
- \(T^a\) is a mapping, \(T^a : L^a \cup L^b \rightarrow \text{Ids}^a\),
- \(\preceq^a\) is a relation in \(L^a \times L^b\), and \(\nabla^a\) is a relation in \(L^b \times L^b\)

and the requirements 1 – 4 are satisfied. The elements of \(L^a \cup L^b\) are called fragments.

We indicate through \(F^a, F^b, F^a_1, \ldots, F^a_n, \text{etc.}\) the fragments belonging to \(L^a\), through \(F^b, F^b_1, F^b_2, \text{etc.}\) the fragments belonging to \(L^b\), and through \(F, F_1, F_2, \text{etc.}\) the fragments belonging to \(L^a \cup L^b\).

Definition 26 Fragments \(F_1, F_2 \in L^a \cup L^b\), are disjoint iff \(T^a(F_1) \cap T^a(F_2) = \emptyset\).

Requirement 1 For any \(F^a \in L^a, F^b \in L^b, F^a \neq \emptyset\):

- \(C^a(\emptyset, F^a) = \emptyset\)
- \(C^a(F^a, F^b) \neq \emptyset \iff F^a \preceq F^b \nabla \implies F^b \preceq F^a \nabla (C^a(F^a, F^b) \nabla)

Requirement 2 For fragments \(F, F' \in L^a \cup L^b, F^a \in L^a, F^b \in L^b\):

- \(T^a(F \cup F') = T^a(F) \cup T^a(F')\)
- \(F = F' \implies \text{or } n = \#(T^a(F)), \exists F_1, \ldots, F_n, F'_1, \ldots, F'_n: F = F_1 + \ldots + F_n, F' = F'_1 + \ldots + F'_n, \text{ and for } i \in \{1 \ldots n\} : F_i = F'_i, \#(T^a(F_i)) = 1\)
- \(F^a \preceq F^b \nabla \implies T^a(C^a(F^a, F^b)) = T^a(F^b)\)

Requirement 3 For fragments \(F^a_1, F^a_2 \subseteq L^a, F^b_1, F^b_2 \subseteq L^b\):

- \(F^a_1 + F^b_2 \implies F^a_1 \text{ and } F^b_2 \text{ disjoint}\)
- \(F^a_1 \preceq F^a_2 \nabla \implies F^a_1 \text{ and } F^a_2 \text{ disjoint}\)

Requirement 4 For fragments \(F^a, F^a_1, F^a_2 \subseteq L^a, F^b, F^b_1, F^b_2 \subseteq L^b\):

- \(F^a \preceq F^a_1, F^a_2 \implies F^a \preceq F^a_1 + F^a_2 \nabla \)
- \(F^a \preceq F^a_1, F^a \preceq F^a_2 \implies F^a_1 \text{ and } F^b_2 \text{ disjoint}\)
- \(F^a_1 \preceq F^a_2 \implies F^a_1 + F^a_2 \preceq F^a \nabla \)
- \(\epsilon \preceq F^a \implies F^a \preceq F^a \nabla \)
- \(F^a_1 \preceq F^b \implies F^a_1 + F^b \preceq F^a \nabla \implies C^a(F^a, F^a_1 + F^b_2) = C^a(F^a, F^a_1 + F^b_2)\)

The exact nature of fragments is language and implementation dependent. In the following sections we study the elements and properties of the fragment system in Java. As an example, we repeat the students example from [9] with source fragments as in figure 30 and binary fragments as in figure 31.
14.2 Elements of a Java fragment system

Definition 27 We define the following sets:

- \( J^a = \{ S | S \) is a Java\(^a \) program as defined in figure 1 \} \)
- \( J^b = \{ B | B \) is a Java\(^b \) program as defined in figure 14 \} \)
- \( J^p = \{ D | D \) is a sequence of descriptions as defined in figure 4 \} \)

Obviously, we aim to use \( J^a \) and \( J^b \) as the source (\( L^a \)) and binary (\( L^b \)) language respectively.

Programs or descriptions can be combined through the linking operator \( _+_- \) to form larger programs or descriptions.

Definition 28 For Java\(^a \) programs \( S_1, S_2 \), Java\(^b \) programs \( B_1, B_2 \) and their descriptions \( D_1, D_2 \) we define the linking operation \( S_1 +_+ S_2 \rightarrow \): \( J^a \times J^a \rightarrow J^a \) \( J^a \times J^b \rightarrow J^b \) \( J^b \times J^b \rightarrow J^b \):

- \( S_1 +_+ S_2 = S_1\text{1} S_2 \)
- \( B_1 +_+ B_2 = B_1\text{2} B_2 \)
- \( D_1 +_+ D_2 = D_1\text{1} D_2 \)

Properties of linked programs and descriptions are given in sections 14.3.

The set of identifiers \( \text{Ids}^d \) stands for all possible identifiers.

Definition 29 \( \text{Ids}^d = \{ \text{id} | \text{id is Identifier} \} \)

The function \( I^d \) extracts the identifiers of all classes and interfaces declared in Java\(^a \) or Java\(^b \) programs.

Definition 30 For Java\(^a \) or Java\(^b \) program \( P \), description \( D \) we define \( I^d : J^a \cup J^b \cup J^p \rightarrow \) \( \text{Ids}^d \):

- \( I^d(e) = \emptyset \)
- \( I^d(\text{class} C \text{ ext} ... \text{ impl} ... \{ \ldots \} P) = \{ C \} \cup I^d(P) \)
- \( I^d(\text{interface} I \text{ ext} ... \text{ impl} ... \{ \ldots \} D) = \{ I \} \cup I^d(D) \)

For the students example as in figures 30, 31, 32 \( I^d(S_{cs}) = I^d(B_{cs}) = I^d(D_{cs}) = \{ \text{CStudent} \} \) and \( I^d(D_{n_a} D_{n_b}) = \{ \text{CStudent, Marker} \} \).

Programs or declarations which introduce different entities are called disjoint.

Definition 31 For descriptions \( D_1, D_2 \) and Java\(^a \) or Java\(^b \) programs \( P_1 \) and \( P_2 \):

- \( D_1, D_2 \) are disjoint iff \( I^d(D_1) \cap I^d(D_2) = \emptyset \)
- \( P_1, P_2 \) are disjoint iff \( I^d(P_1) \cap I^d(P_2) = \emptyset \)

For example, \( D_{n_a} D_{n_b} \) and \( D_{n_a} D_{n_b} \) are disjoint, and \( D_{n_a} D_{cs} \) and \( D_{cs} D_{n_b} \) are not disjoint.

In order to define the compilation of programs in the context of other, imported, programs, we need an update operator \( \oplus \) describing the effect of updating a description with the type information from another one. Updating is associative but not commutative. For disjoint descriptions updating is equivalent to linking.

Definition 32 For descriptions \( D_1, D_2 \)
Figure 30: Student — source fragments

```
// 1st phase
S_{st} = class Student ext Object impl {
    int grade;
};
S_{cs} = class CStudent ext Student impl {};
S_{lab} = class Lab ext Object impl {
    CStudent guy;
    void f() throws { guy.grade=100; }
};
```

```
// 2nd phase
S_{st} = as in 1st phase
S_{cs} = class CStudent ext Student impl {
    char grade;
};
S_{lab} = as in 1st phase
```

```
// 3rd phase
S_{st} = as in 1st and 2nd phase
S_{cs} = as in 2nd phase
S_{lab} = as in 1st and 2nd phase
S_{n} = class Marker ext Object impl {
    CStudent guy;
    void g() throws { guy.grade='A'; }
};
```

- $D_1 \oplus D_2 = D_0 + D_2$, where $D_0$ such that $\exists D_0$ with $D_1 = D_0 + D_3$, and $\mathcal{I}^2(D_0) \subseteq \mathcal{I}^2(D_2)$ and $D_0, D_2$ disjoint

In our example, $\mathcal{I}^2(D_{st}) \subseteq \mathcal{I}^2(D_{cs}D_{lab})$, and $\mathcal{D}(D_{st} + B_{cs} + B_{lab}) \oplus \mathcal{D}(S_{cs} + S_{n}) = D_{st}D_{cs}D_{lab}D_{n}$.

Definition 33: For a Java\(^a\) program S, Java\(^b\) program B we define $\mathcal{C}^2: J^a \times J^b \rightarrow J^b$

- $\mathcal{C}^2(S, B) = \{ C \{ D(B) \oplus D(S), \epsilon, S \} | \mathcal{C} \{ D(B) \oplus D(S), \epsilon, S \} \neq \text{Undefined} \}$

Notice, that we use the non-commutative update operation $D(B) \oplus D(S)$ for the context. This allows S not only to add new type information in the context but also to override the type information, earlier declared in B. For example, this allows to conclude that $\mathcal{C}^2(S_{cs}, B_{st}B_{cs}) \neq \epsilon$.

The predicate $B \uparrow^2 S \Diamond$ represents well-formedness of Java\(^a\) program S in the context of descriptions from Java\(^b\) programs B. The predicate $B \uparrow^b B_1 \Diamond$ represents well-formedness of Java\(^b\) program $B_1$ in the context of descriptions from Java\(^b\) programs B. In both cases the descriptions of B are updated by those from S or $B_1$.

Definition 34: For a Java\(^a\) program S, and Java\(^b\) programs B, $B_1$ we define the relations $\downarrow^b \Diamond$ over $J^b \times J^a$ and $\downarrow^b \Diamond$ over $J^b \times J^b$

- $B \downarrow^b S \Diamond$ iff $(D(B) \oplus D(S)) \uparrow^b S \Diamond$
- $B \downarrow^b B_1 \Diamond$ iff $(D(B) \oplus D(B_1)) \uparrow^b B_1 \Diamond$

Thus, for the example, $D_0 + (D(B_{st} + B_{cs} + B_{lab}) \oplus D(S_{cs} + S_{n})) \uparrow^b S_{cs}S_{n} \Diamond$, and therefore finally $B_{st} + B_{cs} + B_{lab} \uparrow^b S_{cs} + S_{n} \Diamond$. 

78
// 1st phase
B_{st} = class Student ext Object impl {
    int grade;
}
B_{cs} = class CStudent ext Student impl {}
B_{lab} = class Lab ext Object impl {
    Student guy;
    void f() throws { guy.[Student].grade = 100; }
}

// 2nd phase
B_{st} = as in 1st phase
B_{cs'} = class CStudent ext Student impl {
    char grade;
}
B_{lab} = as in 1st phase

// 3rd phase
B_{st} = as in 1st and 2nd phase
B_{cs'} = as in 2nd phase
B_{lab} = as in 1st and 2nd phase
B_{n} = class Marker ext Object impl {
    CStudent guy;
    void g() throws { guy.[CStudent].grade = 'A'; }
}

Figure 31: Students — binary fragments

// 1st phase
D_{st} = class Student ext Object impl {
    int grade;
}
D_{cs} = class CStudent ext Student impl {}
D_{lab} = class Lab ext Object impl {
    CStudent guy;
    void f() throws ;
}

// 2nd phase
D_{st} = as in 1st phase
D_{cs'} = class CStudent ext Student impl {
    char grade;
}
D_{lab} = as in 1st phase

// 3rd phase
D_{st} = as in 1st and 2nd phase
D_{cs'} = as in 2nd phase
D_{lab} = as in 1st and 2nd phase
D_{n} = class Marker ext Object impl {
    CStudent guy;
    void g() throws ;
}

Figure 32: Students — descriptions
14.3 Properties of Java\textsuperscript{a}, Java\textsuperscript{b} programs

Commutativity of linking

In this section we demonstrate that linking is commutative. We show that the compilation and well-formedness of any Java\textsuperscript{a}, Java\textsuperscript{b} programs or descriptions are preserved in whatever order they have been put together.

The descriptions of linked programs are identical to the linked descriptions of the constituent programs.

Well-formedness of a Java\textsuperscript{a} or Java\textsuperscript{b} programs does not depend on the order of descriptions in the context. Compilation of a Java\textsuperscript{a} program does not depend on the order of descriptions either.

Well-formedness of linked Java\textsuperscript{a} programs does not depend on their order. Well-formedness of a Java\textsuperscript{a} program or Java\textsuperscript{b} program in the type information context of linked Java\textsuperscript{b} programs does not depend on their order. Non-empty compilation of linked Java\textsuperscript{a} programs does not depend on their order. Non-empty compilation of a Java\textsuperscript{a} program in the type information context of linked Java\textsuperscript{b} programs does not depend on their order.

**Lemma 13** For Java\textsuperscript{a} or Java\textsuperscript{b} programs P\textsubscript{1}, P\textsubscript{2}

- \(D(P_1 + P_2) = D(P_1) + D(P_2)\)

For descriptions D\textsubscript{1}, D\textsubscript{2}

- \(\forall S \; D_1, D_2 \not\models S \iff D_2 D_1 \not\models S\)
- \(\forall B \; D_1, D_2 \not\models B \iff D_2 D_1 \not\models B\)
- \(\forall S \; C\{D_1, D_2, e, S\} = C\{D_2 D_1, e, S\}\)

For Java\textsuperscript{a} programs S\textsubscript{1}, S\textsubscript{2} and Java\textsuperscript{b} programs B\textsubscript{1}, B\textsubscript{2}

- \(B \not\models S_1 + S_2 \iff B \not\models S_2 + S_1\)
- \(B_1 + B_2 \not\models S \iff B_2 + B_1 \not\models S\)
- \(B_1 + B_2 \not\models B \iff B_2 + B_1 \not\models B\)
- \(C^2\{S_1 + S_2, B\} \neq e \implies \exists B_1, B_2 \text{ such that}\)
  \(C^2\{S_1 + S_2, B\} = B_1 + B_2 \text{ and } C^2\{S_2 + S_1, B\} = B_2 + B_1\)
- \(C^2\{S, B_1 + B_2\} = C^2\{S, B_2 + B_1\}\)

The above lemma means that programs or descriptions consisting of the same parts but put together in different order behave identically in all aspects relevant to our discussion. Therefore the linking operation + is commutative for Java\textsuperscript{a}, Java\textsuperscript{b} programs and for descriptions.

**Lemma 14** For Java\textsuperscript{a} programs S\textsubscript{1}, S\textsubscript{2}, Java\textsuperscript{b} programs B\textsubscript{1}, B\textsubscript{2}, descriptions D\textsubscript{1}, D\textsubscript{2}

- \(S_1 + S_2 = S_2 + S_1\)
- \(B_1 + B_2 = B_2 + B_1\)
- \(D_1 + D_2 = D_2 + D_1\)

Well-formed compilation

Empty programs are compiled into empty programs. Well-formedness of a non-empty Java\textsuperscript{a} program in the context of a Java\textsuperscript{b} program is equivalent to well-formedness of the corresponding non-empty compilation in the context of the same Java\textsuperscript{b} program.

**Lemma 15** For any Java\textsuperscript{a} program B

- \(C^2\{e, B\} = e\)
- \(\forall S \neq e: \; C^2\{S, B\} \neq e \iff B \not\models S \iff B \not\models C^2\{S, B\}\)
Composition and decomposition of programs

Each program consists of components, which are programs as well. Programs are composed into larger programs by the linking operation, if they introduce different program entities, represented by their different sets of identifiers. Simple or minimal programs are those, which introduce one program entity, i.e. class or interface, that is represented by its identifier: \( \#(T^3(P_i)) = 1 \). Compilation of a Java\textsuperscript{a} program introduces the same identifiers as the Java\textsuperscript{b} program.

**Lemma 16** For Java\textsuperscript{a} or Java\textsuperscript{b} programs \( P, P' \), Java\textsuperscript{a} program \( S \), Java\textsuperscript{b} program \( B \):

- \( T^3(P \cdot P') = T^3(P) \cup T^3(P') \)
- \( P = P' \iff \#(T^3(P)) = \#(T^3(P')) \), \( \exists P_i \ldots P_n, P_i' \ldots P_n' \)
  \( P = P_1 \ldots P_n, P' = P_1' \ldots P_n' \), and for \( i \in \{1 \ldots n\} : P_i = P_i', \#(T^3(P_i)) = 1 \)
- \( B \not\in S \implies T^3(C^3(S, B)) = T^3(S) \)

**Lemma 17** For Java\textsuperscript{a} programs \( S_1, S_2 \), and Java\textsuperscript{b} programs \( B, B_1, B_2 \):

- \( B \not\in S_1 + S_2 \implies S_1 \text{ and } S_2 \text{ disjoint} \)
- \( B \not\in B_1 + B_2 \implies B_1 \text{ and } B_2 \text{ disjoint} \)

Program locality

Linking disjoint well-formed Java\textsuperscript{a} or Java\textsuperscript{b} programs produces well-formed programs. If \( B \) is a well-formed Java\textsuperscript{b} program in the context of \( B_1 \), then it remains so in any well-formed larger context \( B_1 + B_2 \). Checking a Java\textsuperscript{b} program in the empty context is equivalent to checking it with itself as the context. Finally, if \( S \) is well-formed in the context of \( B_1 \) and also in the larger context \( B_1 + B_2 \), then compilation in these both environments produces identical results.

**Lemma 18** For Java\textsuperscript{a} programs \( S, S_1, S_2 \) and Java\textsuperscript{b} programs \( B, B_1, B_2 \):

- \( B \not\in S_1 \cup S_2 \iff B \not\in S_1 + S_2 \)
- \( B \not\in B_1 \cup B_2 \iff B \not\in B_1 + B_2 \)
- \( B_1 \not\in B \iff B_1 \not\in B_1 + B_2 \)
- \( \epsilon \not\in B \iff \epsilon \not\in B_1 + B_2 \)
- \( B_1 \not\in S \) and \( B_1 + B_2 \not\in S \iff C^3(S, B_1) = C^3(S, B_1 + B_2) \)

Java\textsuperscript{a}, Java\textsuperscript{b} as a Java fragment system

Thus, we have established all the properties required in definition 25 and thus we obtain a fragment system for Java\textsuperscript{a}, Java\textsuperscript{b} programs.

**Lemma 19** The tuple \( (J^a, J^b, \epsilon, \text{Ids}^a, C^3, T^3, \bullet^a, \bullet^b, \circ, \circ, \circ ) \) is a Java fragment system.

15 Conclusions

We have given a formal description of the operational semantics and type system for a substantial subset of Java including exceptions and exception handling. 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.
To our knowledge, our work is the first to model Java exceptions, and to demonstrate that the type system guarantees that the classes mentioned in the throws-clauses of methods indeed describe any potentially escaping unchecked exceptions.

In [27, 7, 1, 25] operational semantics and type systems for Java and SML exceptions are developed where method types do not mention the exceptions potentially escaping from their bodies. [2] model Java exception handling and virtual machine subroutines, and prove that the compilation of exception related features from Java to JVM is meaning preserving.

The formal system we have developed is very close to Java and to programmers' intuitive ideas about program compilation and execution. So far we have only outlined the proofs of the lemmas and theorem. The work would benefit from the use of a theorem prover, as e.g. in [37, 35].

Modelling Java exceptions and, in particular, the dual role they play whereby they may be used as any object in normal execution until explicitly thrown was quite challenging. Formalization of language features leads to better understanding; for example, the distinction between exception terms and exception-free terms, and their properties, as well as our succinct description of the operational semantics of exception propagation and handling, etc.

On a related topic, the inference of the throws clauses – as opposed to simple checking as in Java – has attracted great interest. The precision of that information has considerable practical implications: An over-cautious programmer, who declares too many exceptions in throws clauses, makes his methods more heavy-weight to call. A tool which analyzes exception flow in Java source code [31], (and, in particular, detects exceptions handled through subsumption) demonstrated that commercial packages tend to handle exceptions at too coarse a grain. A constraint system [41] gives an exception flow analysis algorithm independent of the programmer's specifications. This problem has attracted research in languages where the counterpart of throws clauses does not exist. Control-flow analysis and effects systems have been applied for the estimation of uncaught exceptions in SML programs [42, 30].

Exceptions introduce the potential for non-determinism. For example, the expression e1 and e2 is not equivalent to e2 and e1, because e1 and e2 might raise different exceptions, one of which might be caught. Such considerations could restrict language implementations, i.e. expressions could not be re-arranged unless it can be proven that they raise no exceptions. In [17] a semantics dealing with imprecise exceptions based on the IO monad in Haskell is suggested. This approach is not directly applicable to Java because of the presence of the IO monad, but the issue it tackles is applicable to any language. The question of imprecise exceptions becomes even more pertinent for Java, because the language deliberately under-specifies the time and exact nature of loading and linkage-related errors; although it does place some constraints, e.g. ch. 12.1.2 in [14]. A semantics for this part of the Java language, which would characterize a whole family of non-deterministic implementations and their properties, is an interesting open task.

We have described separate compilation and checking in the context of descriptions which are partially known and are not necessarily well-formed. This is a more faithful account of separate compilation than given in other Java formalizations. We used our description of separate compilation to establish Java\textsuperscript{a} and Java\textsuperscript{b} as a fragment system. This is another validation of the description. A more accurate model of the Java dynamic linker-loader, based e.g. on [39, 16, 20], would complete the picture and is suggested in [6].

Acknowledgments

This work is the result of ongoing research during the last three years. During that period we have received valuable feedback from Peter Sellinger, David von Oheimb, Yao Feng, Sarfraz Khurshid, Guiseppe Castagna, Mark Skipper, David Wragg, Gaurav S. Kc., Elena Zucca, Davide Ancona, and most particularly from Don Syme.
References


84


Index

Demonstration of Java features
  Java program, 32
  Java program, 16
  abnormal type of throw statement, 30
  abnormal type of try-catch-finally statement, 30
  abnormal type, 13
  array access, 7
  array creation, 7
  array types, 7
  array value run-time type checks, 7
  array, 42
  checked exceptions throwable in a method, 9
  compilation of method body, 34
  compilation of program, 12, 34
  compilation Java program, 34
  complete Java program, 39
  conformance for environments, 74
  conformance of state, 74
  covariant array types, 7
  declarations, 18, 78
  difference over abnormal types, 22
  disjoint declarations, 77
  dynamic method binding, 6
  environment, 24
  exception class declaration, 9
  exception declaration in a method header
    – throws clause, 9
  exception propagation, 9
  exceptions as objects, 9
  exceptions thrown and caught within a method, 9
  exceptions, 8
  hiding, 6
  inconsistent sources and linking, 10
  inheritance, 6
  method call operational semantics, 46, 47
  method call with exception throwing and
    catching, 49
  method call with exception throwing and
    propagation, 49
  method call without exception throwing, 49
  more specific abnormal type, 22
  normal type of throw statement, 28
  normal type of try-catch statement, 28
  normal type, 13
  object, 42
  overloading, 6
  overriding, 6
  recursive scopes, 6
  run-time exceptions in the operational
    semantics, 12
  separate compilation and linking without
    errors, 10
  soundness, 75
  state, 11, 42
  static method arguments, 6
  subject reduction, 14
  throwing exceptions, 9
  type-incorrect compilation, 10
  type, 13
  unchecked exceptions, 9
  unpredictable term evaluation, 9
  updating declarations, 78
  value fits type, 46
  well-formed declarations, 18
  well-formed Java program, 39
  well-formed Java programs, 78
  well-formed Java program, 32
  widening over normal types, 14
  widening over types, 14
  ArrayStoreException, 7
  IndexOutOfBoundsException, 7

Examples
  Bcr, 78
  Bcs, 78
  Bshab, 78
  Bn, 78
  Bst, 78
  Scr, 78
  Scs, 78
  Sshab, 78
  Sn, 78
  Sst, 78
  Pcs, 10
  Pcs, 10
  Pshab, 10
  Pn, 10
  Pst, 10
  Bn, 32
  Bph, 32, 34
  Sn, 16
  Sph, 16
  Dcr, 78
  Dcs, 78
  Dshab, 78
  Dn, 78
  Dst, 78
  Db, 18
\begin{align*}
D_{ph} \cdot 18 \\
V_{ph} \cdot 74 \\
V_f \cdot 54 \\
V_{ph} \cdot 24 \\
V_{\var} \cdot 13 \\
P_{at} \cdot 7 \\
P_{\var} \cdot 8 \\
P_{ph} \cdot 5, 6
\end{align*}

**Functions, operations**

\begin{align*}
\tilde{\mathcal{C}}(D, V, t), & 34 \\
\mathcal{C}(D, V, S), & 34 \\
\tilde{\mathcal{C}}(P, \tilde{P}), & 76 \\
\mathcal{C}(S, B), & 12, 78 \\
T(D), & 12, 17 \\
T'(V), & 22 \\
\mathcal{A}(T_{\text{method}}), & 19 \\
\mathcal{A}(T_{\text{method}}), & 31 \\
E_{\text{def}}(T_{\text{method}}), & 19 \\
E_{\text{def}}(T_{\text{method}}), & 19 \\
\mathcal{C}_{\text{def}}(D, C), & 19, 20 \\
\mathcal{C}_{\text{def}}(P, C), & 31 \\
\mathcal{P}_{\text{def}}(D, C, f), & 19, 20 \\
\text{Fields}(D, C, f), & 20, 21 \\
\text{Fields}(P, C, f), & 31 \\
\text{FirstVis}(D, C, f), & 28 \\
\text{IDec}(D), & 19, 20, 31 \\
\text{InitVal}(P, T), & 43 \\
\text{Interf}(D, X, m), & 19, 20 \\
\text{MDec}(D, X, m), & 19, 20 \\
\text{MDec}(P, C, m), & 31 \\
\text{MethId}(T_{\text{method}}), & 19 \\
\text{MethId}(T_{\text{method}}), & 31 \\
\text{Meths}(D, X, m), & 21 \\
\text{Meths}(P, C, m), & 31 \\
\text{Super}(C, D), & 19, 20 \\
\text{Super}(P, C), & 31 \\
\text{ApplMeths}(D, m, T, \text{AT}), & 28 \\
\text{FirstField}(D, m, T, \text{AT}), & 36, 39 \\
\text{FirstField}(P, m, C, \text{AT}), & 31, 32 \\
\text{MostSpec}(D, m, T, \text{AT}), & 28 \\
\end{align*}

**Judgements**

\begin{align*}
\sigma \preceq \sigma', & 53, 54 \\
V \preceq V', & 53, 54 \\
\vdash T \triangleleft_{\text{prim}} \text{tr}, & 46 \\
\vdash D \triangleleft_{u}, & 18-20 \\
\end{align*}
well-formed, \( D' \vdash Dec(D, C) \otimes \), 24
complete, \( D \vdash D \otimes \), 18, 24
disjoint, 77
interface, 17
well-formed, \( D' \vdash I Dec(D, I) \otimes \), 24
of a program, 17
unambiguous, \( D \vdash D \otimes_u \), 18, 19
updating, \( D_u \oplus D_v \), 12, 77
well-formed, \( D' \vdash D \otimes \), 18, 24
different compiled forms of a class, 10
environment, 12, 24, see conformance
well-formed, \( D \vdash V \otimes \), 24
examples, 5
cubs, 7
philosophers, 5
students, 10
worries, 8
exceptions, 8, see operational semantics, exceptions
Exception, 9
Throwable, 8, 9, 17
checked, 8
in C++, 8
in Java, 8
in SML, 8
run-time, 11
ArrayStoreException, 12
NullPointerException, 11
unchecked, 8
Error, 8
RuntimeException, 8
execution
abnormal, 4, 13
normal, 4, 13
expressions, 15, 32, 40, see operational semantics, expressions, method call
this, 15, 24, 32, 40
array access, 15, 32, 40
array creation, 15, 30, 32, 40
field access, 15, 32, 40
method call, 15, 32, 40
object creation, 15, 32, 40
values, 15, 32, 40
field
first visible, \( FirstVis(D, C, t) \), 28
hidden, 21
look-up, see look-up
syntax, 15, 32
fitting, 46, 54
fragment systems, 75
Java as a fragment system, 77–81

array, 41, 42
access to component, 42
new array, 42
class
implements, :\( \text{imp} \), 20
look-up, see look-up
subclass, \( \subseteq \), 20
syntax, 15, 32
well-formed, \( D \vdash C \text{Def}(S, C) \otimes \), 32

compilation, 12, 34
from Java\textsuperscript{a} to Java\textsuperscript{b}, \( C \), 11
of expressions and statements, \( C \), 34
of programs, \( C \), 34
of fragments, \( C^f \), 76
preserves types, 12, 39
separate, \( C^s \), 12, 78
well-formed, 80
separate compilation and linking, 10
configuration, 41
conformance, 53, 54
for environments, \( V \subseteq V' \), 53
for states, \( \sigma \subseteq \sigma' \), 53
of configuration, \( P, V \vdash t, \sigma \triangleleft T \), 53
of state, \( P, V \vdash \sigma \otimes \), 53
of value, \( P, \sigma \vdash \text{val} \triangleleft T \), 53
local, \( P, \sigma \vdash \text{val} \triangleleft \triangleleft T \), 53
description mapping, \( D \), 17
descriptions, 12, 17
acyclic, \( D' \vdash D \otimes_{u} \), 18, 20
properties, 23
basic, \( D_b \), 18
class, 17

Predefined Identifiers
ArrStoreE, 14, 16
Error, 16
Exception, 16
IndOutBndE, 14, 16
NegSizeE, 14, 16
NullPointerException, 14, 16
Object, 14, 16
RTE, 16
Throwable, 16

array, 41, 42
access to component, 42
new array, 42
class
implements, \( \text{imp} \), 20
look-up, see look-up
subclass, \( \subseteq \), 20
syntax, 15, 32
well-formed, \( D \vdash C \text{Def}(S, C) \otimes \), 32

compilation, 12, 34
from Java\textsuperscript{a} to Java\textsuperscript{b}, \( C \), 11
of expressions and statements, \( C \), 34
of programs, \( C \), 34
of fragments, \( C^f \), 76
preserves types, 12, 39
separate, \( C^s \), 12, 78
well-formed, 80
separate compilation and linking, 10
configuration, 41
conformance, 53, 54
for environments, \( V \subseteq V' \), 53
for states, \( \sigma \subseteq \sigma' \), 53
of configuration, \( P, V \vdash t, \sigma \triangleleft T \), 53
of state, \( P, V \vdash \sigma \otimes \), 53
of value, \( P, \sigma \vdash \text{val} \triangleleft T \), 53
local, \( P, \sigma \vdash \text{val} \triangleleft \triangleleft T \), 53
description mapping, \( D \), 17
descriptions, 12, 17
acyclic, \( D' \vdash D \otimes_{u} \), 18, 20
properties, 23
basic, \( D_b \), 18
class, 17

well-formed, \( D' \vdash C \text{Dec}(D, C) \otimes \), 24
complete, \( D \vdash D \otimes \), 18, 24
disjoint, 77
interface, 17
well-formed, \( D' \vdash I \text{Dec}(D, I) \otimes \), 24
of a program, 17
unambiguous, \( D \vdash D \otimes_u \), 18, 19
updating, \( D_u \oplus D_v \), 12, 77
well-formed, \( D' \vdash D \otimes \), 18, 24
different compiled forms of a class, 10
environment, 12, 24, see conformance
well-formed, \( D \vdash V \otimes \), 24
examples, 5
cubs, 7
philosophers, 5
students, 10
worries, 8
exceptions, 8, see operational semantics, exceptions
Exception, 9
Throwable, 8, 9, 17
checked, 8
in C++, 8
in Java, 8
in SML, 8
run-time, 11
ArrayStoreException, 12
NullPointerException, 11
unchecked, 8
Error, 8
RuntimeException, 8
execution
abnormal, 4, 13
normal, 4, 13
expressions, 15, 32, 40, see operational semantics, expressions, method call
this, 15, 24, 32, 40
array access, 15, 32, 40
array creation, 15, 30, 32, 40
field access, 15, 32, 40
method call, 15, 32, 40
object creation, 15, 32, 40
values, 15, 32, 40
field
first visible, \( FirstVis(D, C, t) \), 28
hidden, 21
look-up, see look-up
syntax, 15, 32
fitting, 46, 54
fragment systems, 75
Java as a fragment system, 77–81

D' \vdash D' \otimes_{u} \), 20
D \vdash S \otimes \), 32
D \vdash C \text{Def}(S, C) \otimes \), 32
D \vdash C \otimes \), 20
D \vdash I \otimes \), 20
P, \sigma \vdash \text{val} \triangleleft \triangleleft T \), 53
P, \sigma \vdash \text{val} \triangleleft T \), 53
P, \sigma \vdash \text{val} \triangleleft \triangleleft T \), 53
information, D, 12
interface
look-up, see look-up
subinterface, ≤, 20
syntax, 15, 32
Java, 3–5, 10, 14, 15
Java*, 4–5, 11, 40
context of exception, 41
operational semantics, 41
syntax, 40
type system, 49
Java*, 4–5, 10–12, 14
syntax, 14, 15
type system, 26
Java*, 4–5, 11, 32
type system, 34
link-time checks, 13
linking, +, 77
commutativity of, 80
for Java* programs, 77
for Java* programs, 77
for declarations, 77
look-up
class
in declarations, CDec[D,C], 19
in program, CDec[P,C], 31
direct superclass
in declarations, SuperC[D,C], 19
in program, SuperC[P,C], 31
direct superinterfaces
in declarations, Interfs[D,X], 19
field in class
in declarations, FDec[D,C,f], 19
fields in hierarchy, 20
in declarations, Fields[D,C,f], 20, 21
in program, Fields[P,C,f], 31
interface
in declarations, IDec[D,1], 19
method in class or interface
in declarations, MDec[D,X,m], 19
in program, MDec[P,C,m], 31
methods in hierarchy, 20
in declarations, Meths[D,X,m], 21
in program, Meths[P,C,m], 31
variables or this, V(v), 24

method
throws clause, 17
set of exception classes, ExcT[methH], 19
applicable, ApplMeths[D,m,T,AT], 28
argument type, 16, 17
in declarations, ArgT[methH], 19
in program, ArgT[meth], 31
well-formed, D ⊨ AT ⊓ ArgType, 22
body, 15
well-formed, 32
first fit
in declarations, FirstFit(D,m,T,AT), 36
in program, FirstFit(P,m,C,AT), 31
identifier, 17
in declarations, MethId[methH], 19
in program, MethId[meth], 31
look-up, see look-up
more special, ≺, 28
most special, MostSpec(D,m,T,AT), 28
overridden, 21
return type, 17
in declarations, RetT[methH], 19
well-formed, D ⊨ T ⊓ RetType, 22
syntax, 15, 32
object, 11, 41
access to field, 42
new object, 42
operational semantics, ⇒ 41
assignment, 43, 45
exceptions, 46
expressions, 43
method call, 46
properties, 54
run-time model, 11, 41
statements, 45

primitive types
bool, 15, 32, 40
char, 15, 32, 40
int, 15, 32, 40

program
Java*, 15
basic, S, 16
well-formed, B ⊨ S ⊓, 78
well-formed, B ⊨ S ⊓, 32
well-formed, B ⊨ P ⊓, 31
Java*, 32
basic, B, 32
complete, B ⊨ B ⊓, 39
complete, B ⊨ P ⊓, 39
well-formed, B ⊨ B ⊓, 78
well-formed, B ⊨ P ⊓, 39

programs
composition and decomposition, 81
context, 81
disjoint, 77
soundness theorem, 74
state, 11, 41, 42, see conformance
  access to field by object’s reference, 42
new state, 42
statements, 15, 32, 40, see operational semantics, statements, assignment
  return, 15, 32, 40
  throw, 15, 32, 40
  try, 15, 32, 40
assignment, 15, 32, 40
conditional, 15, 32, 40
method call, 15, 32, 40
subject reduction theorem, 14, 55
subsumption rule, 30

terms
  ground, 42, 43
  1-ground, 42, 43
  value, 43
type systems
  Java\textsuperscript{r}, 49
    properties, 50
  Java\textsuperscript{c}, 26
    properties, 30
  Java\textsuperscript{b}, 34
    properties, 36
types, 16
  in Java\textsuperscript{r}, 49
  in Java\textsuperscript{c}, 26
  in Java\textsuperscript{b}, 34
well-formed, $D \vdash T \parallel ET \diamond_{Type}$, 22
abnormal type, 4, 13, 16, 17
  difference, $\setminus$, 22
  in Java\textsuperscript{r}, 50
  in Java\textsuperscript{c}, 26, 28
  in Java\textsuperscript{b}, 36
more specific, $\subseteq$, 22
well-formed, $D \vdash T \diamond_{AbnType}$, 22
normal type, 4, 13, 16
  $\perp$, 16
  nil, 16
void, 15, 16, 32, 40
  in Java\textsuperscript{r}, 50
  in Java\textsuperscript{c}, 26
  in Java\textsuperscript{b}, 36
variable type, see variable types
well-formed, $D \vdash ET \diamond_{NorType}$, 22
typing, 12
values, 15, see conformance
  initial, 43
  primitive, 16, 32, 40
  reference, 40
  reference null, 15, 16, 32, 40

value fits type, 46
variable types, 15, 32
  array type, 15, 32
  simple type, 15, 32
  class, 15, 32
  interface, 15, 32
  primitive type, 15, 32
well-formed, $D \vdash T \diamond_{VarType}$, 22, 24
widening, $\leq_w$, 14, 23
  over argument types, 23
  over normal types, 14, 23
  over types, 14, 23
  over variable types, 23