Java: Semantics and Type Soundness

Gaurav S. Kc
B. Eng., Department of Computing
Imperial College of Science, Technology and Medicine
e-mail: gsk1@doc.ic.ac.uk

Abstract

Java is a programming language that has promised so much. Yet, those who are cautious are not very willing to use it for all sorts of programming applications. The main problem with Java is that it has borrowed so many features from so many languages and made itself very complicated in the process. This makes it very hard to believe that Java is a very reliable language. And its immense complexity makes study about the language a hard task indeed.

The main approach towards reasoning about a language is by studying its operational semantics, i.e. the meaning of programs written in the language. Drs Drossopoulou and Eisenbach have been heading the research group *SLURP* at Department of Computing, Imperial College for the past few years and they have been deeply involved in research on Java. So far, they have defined an operational semantics for the language, and this semantics also reasons about the type-checking phase of a Java compiler and run-time checking as well – which makes Java as safe as it actually is.

This paper is the report on the research that I have done, and it is based on Drossopoulou and Eisenbach's work. I have extended the definition of their semantics for Java and have made it possible to reason about Java programs having modifiers, constructors, etc.

Contents

List of Figures

1 Introduction

Java is a relatively new general purpose object-oriented language which has become very popular and come into widespread use. It is unique in that it includes a number of features from many languages, most important of which are Smalltalk and C++. However, this inclusion of so many features makes it is more difficult to be able to accurately reason about the behaviour of programs written in Java. This is probably the most important reason why Java has not yet been used in mission-critical programming projects where reliability of the programs cannot be compromised. It has also not yet been possible to guarantee the thoroughly safe behaviour of Java programs. The problem has, fortunately, not escalated to high proportions because the Java designers did not include new and untested features in the language design – all the *imported* features were already well-known and tested in the other languages. But then again, the very complex combination of programming language features that Java incorporates has never existed before in any single language. The features occurred in isolation among these other languages where testing could be carried out relatively easily. to test those features. Drossopoulou and Eisenbach had developed a [small-step] operational semantics in [1] for a substantial subset of Java, and so gone on to prove that as far as that subset was concerned, Java's type system was sound. In the summer of 1998, I was working on a UROP project that was based in its entirety on the work done in [1] . My task then was to build a tool that would:

- Implement the semantics rules in [1] so as to provide some sort of simulator environment where the execution of test Java programs could be observed, and
- Automate much of the tedious theorem-proving process that goes on when proving the type-soundness of programming languages.

This research is a derivation from the afore-mentioned UROP project in the sense that I have extended the subset of Java that is covered by the semantics. In other words, I have derived semantics rules for those aspects of the language which were yet uncovered: viz. access modifiers, final modifier and constructors.

2 Java: So Far

The key aspects of Java covered by the semantics in [1] that are relevant to this paper are classes and interfaces, inheritance of instance variables and instance methods, non-inheritable static variables, overshadowing of instance variables and overriding of instance methods, overloading of methods, representation of class hierarchies and type declarations in the environment, and representation of program execution in the state.

The main idea of proving type-soundness is that some Java program of a certain type [void] would be compiled to a Java_{se} program of the same type thus ensuring type-soundness. If, on the other hand, a compilation error arose, the compilation would halt and produce an error message and no files would be output. At each point during execution, the absence of a runtime error (exception) could also be used to show that the type of the program was always either unchanged or made more specific. Again, on the other hand, an erroneous situation occurring would definitely raise an exception and stop execution, hence preventing any changes to the state. This means that the behaviour of the program was always as expected. This two-phase type-verification and checking for errors and erroneous behaviour in Java is what maintains and checks the integrity and type-soundness of the Java programs.

In [1] Java_s, a provably safe subset of Java was defined. Java_s contains primitive types, classes and inheritance, instance variables and instance methods, interfaces, shadowing of instance variables , dynamic methods binding, object creation, the `null` value, arrays, exceptions and exception handling. Java_s was then enriched with compile-time type information required for Java program execution, resulting in Java_{se} . This type information is needed for resolving fields and methods of objects , clases and interfaces as well as constructors of classes. The enriched version, Java_{se}, is only the first step in the execution of the Java programs. During the actual execution, Java_{se}terms can be re-written to such terms that cannot be classified as proper Java_{se} terms without extending the Java_{se} syntax. These situations arise because of the need to represent the working memory of the JVM during run-time. So, Java_r was declared in [1] – Java_r is a superset of Java_{se} in the sense that it can have all the terms possible in Java_{se} as well as those run-time terms pertaining to the actual state information required during program execution.

Terms in each of the three subsets of Java are typed terms. The type of a term comprises of its *normal* type and its *abnormal* type. What this means is that the type system covers for both cases when

- the execution of a program is normal and as expected with all thrown exceptions being caught and dealt with and
- the term in question throws an exception which isn't caught

For any term in any of these three systems, its *normal* term will be one of the defined classes or interfaces, or some primitive type like `nil`, `void`, `bool`, `int`, etc. Also, its *abnormal* type will be a set of the subclasses of the predefined Exception class.

$$\begin{array}{ccccccccc}
\text{Java} & \supset & \text{Java}_s & \underset{\mathcal{C}}{\longrightarrow} & \text{Java}_{se} & \subset & \text{Java}_r & \underset{\mathtt{p}}{\leadsto} & \text{Java}_r \\
 & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 & & \text{Type} & = & \text{Type} & = & \text{Type} & \geq_w & \text{Type}
\end{array}$$

The type of any Java_s term is preserved by the enriching phase so that the version in Java_{se} has exactly the same type. Also, any Java_{se} term will have the same type during execution in Java_r as in Java_{se} for all possible machine states.

There is a subject reduction theorem that has been proved in [1], and this theorem describes how in *normal* execution of a program, the *normal* types are preserved, and in case of *abnormal* exectution, an exception results. This exception will be a subclass of some exception class as was defined in the term's *abnormal* type. So, there are some predictions that can be made during the execution of programs, viz. even in case of exceptions, the set of possible exceptions can be pre-determined.

2.1 Basic key features of Java: a brief description

The following features of Java have been captured in the semantics so far:

- Recursive scope
  A class / interface is visible outside its declaration.
- Variable overshadowing
  declaration of a variable in a subclass using an identifier already used for a variable declaration.
- Method overriding
  The method version of overshadowing ... using the same method identifier in a subclass.
- Method overloading
  Multiple declarations of the same method in the same class or subclass with mutually unique signatures, whether through different number of arguments, or different argument types, or even different ordering of the same number of the same arguments.
- Class extension
  when a class is a subset of another, the subset class inherits all its parent class' declared methods and variables along with the ones that the parent class had inherited.
- Interface extension
  when an interface extends any number of other interfaces, it inherits all of their [implicitly **public static final**] fields along with the [**abstract**] method declarations.
- Interface implementation
  when a class implements any number of interfaces, it inherits all their fields and methods. This means that the class has to provide implementations for all the methods it inherits from the interfaces!

2.2 Illustration of Java features: Original Semantics

```
class Animal extends Object f
    boolean isAlive ;
    boolean isHealthy ;
    boolean attack(Cat c) f ...g
g

classs Bird extends Animal f ...g

class Cat extends Animal implements Carnivore f
    int isAlive ;
    void attack(Cat c) f ...g
    void attack(Dog d) f ...g
    void eat(Bird b) f ...g
g

class Dog extends Animal f ...g

interface Carnivore f
    void eat(Bird b) ;
g
```

The program above demonstrates the following features of Java:

- Recursive scope
  Class `Dog` is visible inside class `Cat` even though their declarations do not mutually encompass in any way.
- Variable overshadowing
  The boolean `isAlive` variable in class `Animal` is overshadowed by the integer variable declared in class `Cat` with the same name.
- Method overloading/overriding
  The method `attack()` in class `Animal` is overridden by a re-declaration of the method with the same signature in class `Cat`. The `attack` methods in class `Cat` overload each other because they have the same identifier and the only difference in their declarations lies in their signatures.
- Class/Interface extension
  Class `Cat` extends superclass `Animal`
- Interface implementation
  Class `Cat` implements interface `Carnivore` and inherits its `eat` method
- Method/Field inheritance
  Class `Cat` inherits the `isHealthy` variable from its parent class `Animal`

3 Java Semantics: An Extension — Constructors, Class and Interface members and Modifiers

3.1 Extended key features of Java: a brief description

What is new in the semantics?

The formal definition of the operational semantics for Java now covers a much larger subset of the programming language. The inclusion of the semantics rules defining the effect of access modifiers makes it possible to specify the accessibility of class members and constructors. It is only now that the concept of security through information hiding can be modelled because of the feasibility of accurately simulating the behaviour of restricted-access methods and fields of classes. On the other hand, through the use of **final** declarations for fields and methods, it is also easy to model the constraints that prohibit overshadowing of fields and overriding of methods in subclasses.

In practice, space for objects is allocated dynamically at run-time just before the constructor is invoked. The original semantics did not model this, and instead, the whole object space was set up in the Java_{se} phase and not during execution as should have been the case. The extended semantics attempts to model this as closely to the actual behaviour of the JVM as possible – this involves allocating space for all the fields [inherited and declared] of the object in one go, right before the first constructor is invoked, or halting during execution in case there is not enough free space on the program heap. The intricate process of heirarchical and lateral invocation of constructors is also studied to determine the exact order in which initialisation of the instance variables of each class in the subclass hierarchy takes place.
Also, the inclusion of **static** fields in classes and interfaces dictates that the representation of the state be extended. The run-time configuration now not only has entries for objects and arrays, but also for every class and interface defined in the program. Each class and interface will have space for its **static** fields in the state. In case of interfaces, all the fields are **static** fields, and so all the fields defined in an interface will be in that interface's entry.

The following are the additions to the semantics:

- access modifiers for methods and fields of classes and interfaces
- **final** modifiers for non-overrideable methods of classes
- **static** and **final** modifiers and initialisers for fields of interfaces
- **static** modifiers for fields of classes
- **final** modifiers and initialisers for fields of classes
- constructors for classes
- **abstract** and **final** modifiers for non-instantiable and non-extendable classes respectively

3.2 Illustration of Java features: Extended Semantics

The following Java program demonstrates the various features of Java studied:

$P_{\texttt{Animal}} =$

```
abstract class Animal extends Object {
    protected boolean isHealthy ;
    Animal() {
        super();
    }
    Animal( boolean healthy ) {
        this() ;
        this.isHealthy=healthy ;
    }
    void move() {...}
    protected abstract boolean health() ;
}

final class Cat extends Animal implements Carnivore {
    public int numLives = 9 ;
    static int numCatsAlive ;
    public void attack(Cat c) {...}
    private void attack(Dog d) {...}
    public void eat(Cat c) {
        Cat.numCatsAlive = Cat.numCatsAlive - 1 ;
    }
    void move() {...}
    public boolean health() {...}
}

class Dog extends Animal {
    public final int numLegs = 4 ;
    public int siblings ;
    public boolean isHealthy = true ;
    Dog() {
        this(0) ;
    }
    private Dog( int sib ) {
        this(true);
        this.siblings = sib ;
    }
    Dog( boolean healthy ) {
        super(healthy) ;
        this.isHealthy = healthy ;
    }
    public boolean health() {...}
}

interface Carnivore {
    public static final Cat diet = new Cat() ;
    public abstract void eat(Cat c) ;
}
```

This example demonstrates:

- Accessibility, *e.g.* in class `Dog` the instance variable `isHealthy` is declared **public**, and therefore is visible outside class `Dog`, wherever `Dog` is visible.
- Non-accessibility, *e.g.* as accessing the **private** instance method **void** `attack(Dog)` from outside the class `Cat` is illegal.
- Variables depicting behaviour typical of **constants**, as any attempt to assign to the **final** instance variable `numLegs` even in the class `Dog` would result in a compilation error.
- Variable shared between any number of instances of a class `Cat` as is exemplified by **static** variable `numCatsAlive`.
- Prevention of instantiation of a class by declaring it **abstract** as it is done with class `Animal`.
- Prevention against subclassing of class `Cat` by declaring it **final**.
- If desired so, deferred implementation of some methods of a class can be achieved as by declaring method **boolean** `health()` in class `Animal` as **abstract**, and then providing the actual [varying] implementations for the method in each of the different subclasses, viz. `Cat` and `Dog`.
- Explicit (**super**) constructor invocation, *e.g.* **super()** is the first explicit call in `Dog`(**boolean** `healthy`).
- Explicit (**this**) constructor invocation, as **this(0)** is the first explicit call in `Dog`().

3.3 Java$_s$ – extended

As defined in [1] Java$_s$ included classes, interfaces, instance variables and methods, inheritance of instance variables and methods, shadowing of instance variables, overriding and overloading of instance methods, object creation and access, arrays, array creation, several exception classes, etc. This extended version of Java$_s$ incorporates initialisers, constructors, [**static**] class and interface variables, access modifiers for classes, interfaces, fields, methods and constructors, as well as other modifiers like **final** and **abstract** for classes and methods.

Java$_s$ has been re-designed to represent the actual syntax of Java as closely as possible. However, with a view to have a simpler description, the semantics of Java$_s$ imposes some limitations that are otherwise trivial or even non-existent in Java. Some of these are:

- the implicit superclass `Object` has to be explicitly specified in Java$_s$
- the **impl** [implements] and **throws** clauses are no longer optional
- constructors have to have at least one explicit constructor invocation statement, i.e. `this(..)` or `super(...)`
- classes have to have at least one constructor – the compiler will not supply one automatically

```
P_FUIO/s =  class Object f ...g
            class Exception ext Object f ...g
            class RTE ext Exception f ...g
            class NullPE ext RTE f ...g
            class ArrStoreE ext RTE f ...g
            class IndOutBndE ext RTE f ...g
            class NegSzeE ext RTE f ...g ...
```

Figure 1: The Java$_s$ fundamental program

The fundamental program $\mathtt{P}_{\mathtt{FUIO}/s}$, defined in figure 1, is a particular Java$_s$ program that contains the basic classes and interfaces described in chapters 20-22 of [8]. Particularly, for our current work it includes the exception class `Exception`, the runtime exception class `RTE` with the subclasses `NullPE`, `ArrStoreE`, `IndOutBndE` and `NegSzeE`.

Figure 2 describes what a Java$_s$ program looks like. A Java$_s$ program is a sequence of class or interface definitions. Each class definition consists of the modifier(s) of the class followed by the class identifier, its [explicit] parent class, the interfaces the class implements and the body of the class, which has all its constructor and member definitions. Similarly, an interface definition also lists the modifiers and member definitions of the interface along with all the other interfaces extended by the interface in question.

Each definition of a constructor for a class consists of the header and the body, as is the case for all non-**abstract** methods of classes. The body of constructors and methods comprises of a block of statements, while the header is made up of the modifiers, the constructor or method identifier, the identifiers and types of the formal parameters and the list of exception classes in its **throws** clause. An additional part of a method header definition is its return type. The first statement in a constructor is an explicit constructor invocation statement – this is an invocation of either another constructor of the same class, or a constructor of the superclass. All method and constructor bodies are required to end in a **return** statement.

Members of classes and interfaces can be fields too. The definition for these consist of the modifiers for the field, its identifier, type and an optional [mandatory for **final** fields] initialising expression.

Any Java$_s$ statement block will consist of statements which will be either a conditional statement, an assignment, a method invocation, or a **try** or a **throw** statement. **For** and **while** loops

Program	::=	*Def**
Def	::=	*ClassMod* **class** `ClassId` **ext** `ClassName` **impl** `InterfName`*
		f *Constructor** *ClassMember**g
	|	*InterfMod* **interface** `InterfId` **ext** `InterfName`*
		f *InterfMember**g
ClassMod	::=	[**public**] [**abstract**| **final**]
InterfMod	::=	[**public**] **abstract**
AccessMod	::=	**public** | **protected** | **private** | ϵ
ConstrMod	::=	*AccessMod*
FieldMod	::=	*AccessMod* [**static**] [**final**]
IFieldMod	::=	**public static final**
MethMod	::=	*AccessMod* [**final**| **abstract**]
Constructor	::=	*ConstrHeader* *ConstrBody*
ConstrHeader	::=	*ConstrMod* `ConstrId`((*VarType*`ParId`)*)
		throws `ClassName`*
ConstrBody	::=	f *ConstrInvoke*; *Stmts*; **returng**
ConstrInvoke	::=	`this` (*Expr**) | `super` (*Expr**)
ClassMember	::=	*Field* | *Method*
InterfMember	::=	*IField*
Field	::=	*FieldMod* *VarType* `VarId` [*Expr*] ;
IField	::=	*IFieldMod* *VarType* `VarId` *Expr* ;
Method	::=	*MethHeader* [*MethBody*]
MethHeader	::=	*MethMod* (**void** | *VarType*) `MethId` ((*VarType* `ParId`)*)
		throws `ClassName`*
MethBody	::=	f *Stmts* ; **return** [*Expr*] g
Stmts	::=	ϵ | *Stmts* ; *Stmt*
Stmt	::=	**if** *Expr* **then** *Stmts* **else** *Stmts*
	|	*Var* = *Expr* | *Expr*.`MethName` ([*Expr**]) | **throw** *Expr*
	|	**try** *Stmts* (**catch** `ClassName Id` *Stmts*)* **finally** *Stmts*
	|	**try** *Stmts* (**catch** `ClassName Id` *Stmts*)$^+$
Expr	::=	*Value* | *Var* | *Expr*.`MethName` (*Expr**)
	|	**new** `ClassName` (*Expr**) | **new** *SimpleType* ([*Expr*])$^+$([])*
Var	::=	`Name` | *RecvExpr*.`VarName` | *Var* [*Expr*] | **this**
RecvExpr	::=	*Expr* | `ClassName` | `InterfaceName`
Value	::=	*PrimValue* | `null`
PrimValue	::=	*intValue* | *charValue* | *boolValue* | ...
VarType	::=	*SimpleType* | *ArrayType*
SimpleType	::=	*PrimType* | `ClassName` | `InterfaceName`
ArrayType	::=	*SimpleType* [] | *ArrayType* []
PrimType	::=	**bool** | **char** | **int** | ...

Figure 2: Java$_s$ programs

and **case** statements have not been included in the semantics since these can be recoded using conditionals and recursive method invocation.

Java$_s$ expressions can be values, method invocations, and variable accesses. All values are primitive or references [possibly **null**] to objects and arrays. The expression **new** C $(e_1...e_n)$ creates a new object of class C, whereas the expression **new** $\texttt{T[e}_1\texttt{]...[e}_n\texttt{][]}_1\texttt{...[]}_k$, $\texttt{n} \geq 1, k \geq 0$ creates a n+k-dimensional array value.

The Java animal program $\texttt{P}_{\texttt{Animal}}$ corresponds to the following Java$_s$ program

$\texttt{P}_{\texttt{Animal}/s}$ =

```
abstract class Animal  ext Object  impl ∅ f
    protected boolean isHealthy ;
    Animal()  throws  f /* animalConstrBody1 */ g
    Animal( boolean healthy )  throws  f /* animalConstrBody2 */ g
    void move()  throws  f /* animalMoveBody1 */ g
    protected abstract boolean health()  throws  ;
g

final class Cat  ext Animal  impl Carnivore f
    public int numLives ;
    static int numCatsAlive ;
    public void attack(Cat c)  throws  f /* attackBody1 */ g
    private void attack(Dog d)  throws  f /* attackBody2 */ g
    public void eat(Cat c)  throws  f /* eatBody1 */ g
    void move()  throws  f /* catMoveBody1 */ g
    public boolean health()  throws  f /* catHealthBody1 */ g
g

class Dog extends Animal  impl ∅ f
    public final int numLegs ;
    public int siblings ;
    public boolean isHealthy ;
    Dog()  throws  f /* dogConstrBody1 */ ...g
    private Dog( int sib )  throws  f /* dogConstrBody2 */ g
    Dog( boolean healthy )  throws  f /* dogConstrBody3 */ g
    public boolean health()  throws  f /* dogHealthBody1 */ g
g

interface Carnivore  ext ∅ f
    public static final Cat diet;
    public abstract void eat(Cat c)  throws  ;
g
```

3.3.1 Metatypes

Metatypes, defined in figure 3, correspond to type notions of the language Java, which however do not appear as such in Java programs. The metatypes are *type*, *normal type*, *abnormal type*, *argument type*.

*Argument type*s describe the types of arguments required in a method header. *Argument type*s are tuples formed of *variable type*s, which may be primitive types, interfaces, classes or arrays as defined in figure 2.

*Normal type*s indicate the type of possible results of term under the *normal execution*. A *normal type* may be a *variable type*, or nil or **void**.

Type	::=	*NormType* ‖ *AbnormType*
	|	`ClassName-Thrn`
NormType	::=	*RetT()* | `nil`
AbnormType	::=	{ `ClassName` (, `ClassName`)* }
RetT()	::=	*VarType* | **void**
ArgType	::=	[*VarType* (× *VarType*)*]

Figure 3: Metatypes

*Abnormal type*s describe the exceptions escaping from a method under *abnormal execution* and which are required in a method header. An *abnormal type* consists of a set of subclasses of the `Exception` class.

A *type* is either a combination of a *normal type* and an *abnormal type* (`T` ‖ `ET`) or a thrown exception type (`E-Thrn` where `E` is an exception class). The thrown exception types appear only during execution.

3.3.2 Declarations

Declarations	::=	*Declaration**
Declaration	::=	`classMod` **class** `ClassId` **ext** `ClassName` **impl** `InterfName`*
		f *ConstrSign** *CMemSign**g
	|	`interfMod` **interface** `InterfId` **ext** `InterfName`*
		f *IMemSign**g
ConstrSign	::=	*ConstrHeader*
CMemSign	::=	`fieldMod T` *Field* | *MethHeader*
IMemSign	::=	`fieldMod T` *Field* | *MethHeader*

Figure 4: Declarations

The fact that Java permits separate compilation of source files makes it important to be able to extract type information from those very units of compilation. Such information is extracted as the declarations of the compilation units. The declaration of programs consists of the declarations of each of its classes and interfaces, which in turn comprise of their class or interface identifier, its superinterfaces, all its method headers including method identifier, parameters, return type, argument type and its abnormal type. In addition to all this, the declarations of classes also include their superclasses and constructor headers with parameters, and argument and abnormal types.

$\mathtt{D_{FUIO}}$=	**class** `Object` f ...g
	class `Exception` **ext** `Object` f ...g
	class `RTE` **ext** `Exception` f ...g
	class `NullPE` **ext** `RTE` f ...g
	class `ArrStoreE` **ext** `RTE` f ...g
	class `IndOutBndE` **ext** `RTE` f ...g
	class `NegSzeE` **ext** `RTE` f ...g ...

Figure 5: The fundamental declarations

The declarations $\mathtt{D_{FUIO}}$ of the fundamental program $\mathtt{P_{FUIO}}$ is defined in figure 5 and it shows the declarations of the classes and interfaces of $\mathtt{P_{FUIO}}$.

Declarations are mapped from both Java$_s$and Java$_{se}$programs using the mapping $\mathcal{D}$ as follows:

$$\mathcal{D} : \text{Java}_s\text{-Program} \longrightarrow \text{ Declarations } \cup \text{ Java}_{se}\text{-Program} \longrightarrow \text{ Declarations}$$

$\mathcal{D}$ can be naturally extended to any composition of Java$_s$ or/and Java$_{se}$ programs:

$$\mathcal{D} : (\text{Java}_s\text{-Program}^* \text{ Java}_{se}\text{-Program}^*)^* \longrightarrow \text{ Declarations}$$

Definition 1 *For class and interface definitions, the mapping $\mathcal{D}$ is defined in the following way:*

$\mathcal{D}$(`interfMod` **interface** `I` **ext** `I`$_1$,...,`I`$_j${ `iMem`$_1$...`iMem`$_k$}) =
  `interfMod` **interface** `I` **ext** `I`$_1$,...,`I`$_j${ $\mathcal{D}$(`iMem`$_1$)...$\mathcal{D}$(`iMem`$_k$)}

$\mathcal{D}$(`classMod` **class** `C` **ext** `C'` **impl** `I`$_1$,...,`I`$_j${ `constr`$_1$...`constr`$_j$; `cMem`$_1$...`cMem`$_k$}) =
  `classMod` **class** `C` **ext** `C'` **impl** `I`$_1$,...,`I`$_j${ $\mathcal{D}$(`constr`$_1$)...$\mathcal{D}$(`constr`$_j$); $\mathcal{D}$(`cMem`$_1$)...$\mathcal{D}$(`cMem`$_k$)}

$$\mathcal{D}(\texttt{cMem}) = \begin{cases} \texttt{mod T f;} \quad \textit{iff} \quad \texttt{cMem} = \texttt{mod T f Expr;} \\ \texttt{methH} \quad \textit{iff} \quad \texttt{cMem} = \texttt{methH mBody}, \quad \textit{where} \ \texttt{methH} \ \textit{is a method header} \\ \textit{Undef} \quad \textit{otherwise} \end{cases}$$

$$\mathcal{D}(\texttt{constr}) = \texttt{constrH} \quad \textit{iff} \quad \texttt{constr} = \texttt{constrH constrBody}, \quad \textit{where} \ \texttt{constrH} \ \textit{is a constructor header}$$

$$\mathcal{D}(\texttt{iMem}) = \begin{cases} \texttt{mod T f;} \quad \textit{iff} \quad \texttt{iMem} = \texttt{mod T f Expr} \\ \texttt{methH} \quad \textit{iff} \quad \texttt{iMem} = \texttt{methH}\ , \ \textit{where} \ \texttt{methH} \ \textit{is a method header} \\ \textit{Undef} \quad \textit{otherwise} \end{cases}$$

The declaration of Java animals program $\texttt{P}_{\texttt{Animal}/s}$ from is the following:

$\texttt{D}_{\texttt{Animal}} = \mathcal{D}(\texttt{P}_{\texttt{Animal}/s})$ =

```
abstract class Animal ext Object impl ∅ {
    protected boolean isHealthy ;
    Animal() throws
    Animal( boolean healthy ) throws
    void move() throws
    protected abstract boolean health() throws ;
}

final class Cat ext Animal impl Carnivore {
    public int numLives ;
    static int numCatsAlive ;
    public void attack(Cat c) throws
    private void attack(Dog d) throws
    public void eat(Cat c) throws
    void move() throws
    public boolean health() throws
}

class Dog extends Animal impl ∅ {
    public final int numLegs ;
    public int siblings ;
    public boolean isHealthy ;
    Dog() throws
    private Dog( int sib ) throws
```

```
    Dog( boolean healthy )  throws
    public boolean health()  throws
g

interface Carnivore  ext ∅ f
    public static final Cat diet;
    public abstract void eat(Cat c)  throws
g
```

Properties about program entities are deduced using the complete declaration $\mathtt{D} = \mathcal{D}(\mathtt{P})$ of a whole program P, thus enabling shared knowledge about Java program entities – the *basis for Java programming*.

3.3.3 Establishing well-formedness of Java programs

A Java program P is well-formed, *i.e.* $\vdash \mathtt{P} \diamond$, iff for $\mathtt{D} = \mathcal{D}(\mathtt{P})$

- $\vdash \ \mathtt{D} \diamond_u$, *i.e.* declarations are unambiguous
- $\vdash \ \mathtt{D} \diamond_a$, *i.e.* the class and interface hierarchies are acyclic,
- $\vdash \mathtt{D} \diamond$, *i.e.* the declarations are well-formed,
- $\vdash \mathtt{P} \diamond$, *i.e.* the method bodies are well-formed and return values (or throw exceptions) as declared in their headers.

These requirements may be established in the following consecutive phases.

1. Declarations are unambiguous if every class, interface and field has a unique declaration, as may be established according to the rules in figure 6. (Methods are not unique in Java: these can be overloaded; the requirements for a method to be well-formed are given in the corresponding rules for class and interface in figures 10 and 12.) For unambiguous declarations we define various look up functions in section 3.3.7. These functions are used for the establishing well-formedness of class and interface declarations in figure 10.

2. For declarations D satisfying $\vdash \ \mathtt{D} \diamond_u$, lack of cycles in the class and interface hierarchies may be established according to the rules from figure 7 in section 3.3.5. For D with $\vdash \ \mathtt{D} \diamond_a$ the functions which look-up all class and interface components are well-defined. These functions are necessary for the well-formed class and interface declarations described in section 3.3.7 and for the typing, as described in sections 3.6, 3.8.

3. For D with $\vdash \ \mathtt{D} \diamond_a$ well-formedness of the class, interface and program specifications can be established according to the rules in figure 10 in section 3.3.7.

4. For a program P with $\vdash \mathcal{D}(\mathtt{P}) \diamond$ we can establish well-formedness of method bodies according to the rule in figure 12 of section 3.3.11.

3.3.4 Unambiguous declarations

Figure 6 describes the requirements that make declarations unambiguous. They expect outer level entities (*i.e.* classes or interfaces) and fields in classes to be unique.

For D with $\vdash \ \mathtt{D} \diamond_u$ the functions $\mathit{ISpec}(\mathtt{D},\mathtt{I})$, $\mathit{CSpec}(\mathtt{D},\mathtt{C})$, $\mathit{FSpec}(\mathtt{D},\mathtt{C},\mathtt{f})$, $\mathit{MSpec}(\mathtt{D},\mathtt{X},\mathtt{m})$ look up the declarations of interface I, class C, field f and method m respectively. The functions $\mathit{SupCSpec}(\mathtt{D},\mathtt{C})$, $\mathit{Interfs}(\mathtt{D},\mathtt{X})$ look up the direct superclass of the class C and all direct superinterfaces of X while the function $\mathit{ConstrSpec}(\mathtt{D},\mathtt{C})$ looks up the superclasses of class C.

$$
\begin{array}{l}
\forall \mathtt{I} \quad \mathtt{D} = \mathtt{D}_1\ \mathtt{interfMod}\ \mathbf{interface}\ \mathtt{I}\ \ \mathbf{ext}\ \mathtt{I}_1, \ldots \mathtt{I}_\mathtt{n} \mathrm{f}\ \ldots\ \mathrm{g}\ \mathtt{D}_2, \\
\qquad \mathtt{D} = \mathtt{D}_3\ \mathtt{interfMod}\ \mathbf{interface}\ \mathtt{I}\ \ \mathbf{ext}\ \mathtt{I}'_1, \ldots \mathtt{I}'_\mathtt{n} \mathrm{f}\ \ldots\ \mathrm{g}\ \mathtt{D}_4 \\
\qquad \Longrightarrow\ \ \mathtt{D}_1 = \mathtt{D}_3,\ \mathtt{D}_2 = \mathtt{D}_4\ \ \text{and} \\
\qquad\qquad \mathtt{D} \neq \mathtt{D}_5\ \mathtt{classMod}\ \mathbf{class}\ \mathtt{I}\ \mathbf{ext}\ \ldots \mathrm{f}\ \ldots\ \mathrm{g}\ \mathtt{D}_6 \\
\forall \mathtt{C} \quad \mathtt{D} = \mathtt{D}_1\ \mathtt{classMod}\ \mathbf{class}\ \mathtt{C}\ \ \mathbf{ext}\ \ \mathtt{C}'\ \mathbf{impl}\ \mathtt{I}_1, \ldots \mathtt{I}_\mathtt{j} \mathrm{f}\ \ldots\ \mathrm{g}\ \mathtt{D}_2, \\
\qquad \mathtt{D} = \mathtt{D}_3\ \mathtt{classMod}\ \mathbf{class}\ \mathtt{C}\ \ \mathbf{ext}\ \ \mathtt{C}''\ \mathbf{impl}\ \mathtt{I}'_1, \ldots \mathtt{I}'_\mathtt{j} \mathrm{f}\ \ldots\ \mathrm{g}\ \mathtt{D}_4 \\
\qquad \Longrightarrow\ \ \mathtt{D}_1 = \mathtt{D}_3,\ \mathtt{D}_2 = \mathtt{D}_4 \\
\forall \mathtt{f} \quad \mathtt{D} = \mathtt{D}_1\ \mathtt{classMod}\ \mathbf{class}\ \mathtt{C}\ \ \mathbf{ext}\ \ \mathtt{C}'\ \mathbf{impl}\ \mathtt{I}_1, \ldots \mathtt{I}_\mathtt{j} \\
\qquad\qquad\qquad \mathrm{f}\, defs_1\ \mathtt{fMod}\ \mathtt{T}\ \mathtt{f};\ defs_2\ \mathrm{g}\ \mathtt{D}_2, \\
\qquad \mathtt{D} = \mathtt{D}_1\ \mathtt{classMod}\ \mathbf{class}\ \mathtt{C}\ \ \mathbf{ext}\ \ \mathtt{C}'\ \mathbf{impl}\ \mathtt{I}_1, \ldots \mathtt{I}_\mathtt{j} \\
\qquad\qquad\qquad \mathrm{f}\, defs_3\ \mathtt{fMod}'\ \mathtt{T}'\ \mathtt{f};\ defs_4\ \mathrm{g}\ \mathtt{D}_2 \\
\qquad \Longrightarrow\ \ defs_1 = defs_3,\ defs_2 = defs_4 \\
\forall \mathtt{f} \quad \mathtt{D} = \mathtt{D}_1\ \mathtt{interfMod}\ \mathbf{interface}\ \mathtt{I}\ \ \mathbf{ext}\ \mathtt{I}_1, \ldots \mathtt{I}_\mathtt{n} \\
\qquad\qquad\qquad \mathrm{f}\, defs_1\ \mathtt{fMod}\ \mathtt{T}\ \mathtt{f};\ defs_2\ \mathrm{g}\ \mathtt{D}_2, \\
\qquad \mathtt{D} = \mathtt{D}_1\ \mathtt{interfMod}'\ \mathbf{interface}\ \mathtt{I}\ \ \mathbf{ext}\ \mathtt{I}_1, \ldots \mathtt{I}_\mathtt{n} \\
\qquad\qquad\qquad \mathrm{f}\, defs_3\ \mathtt{fMod}'\ \mathtt{T}'\ \mathtt{f};\ defs_4\ \mathrm{g}\ \mathtt{D}_2, \\
\qquad \Longrightarrow\ \ defs_1 = defs_3,\ defs_2 = defs_4 \\
\hline
\vdash\ \mathtt{D} \diamond_u
\end{array}
$$

Figure 6: Unique declarations

In order to extract the method declaration components, namely, a method identifier, *argument type*, *return type*, *abnormal type* and modifiers from a method declaration we introduce the functions $MethId(\_)$, $ArgT(\_)$, $RetT(\_)$, $ExcT(\_)$ $Mod(\_)$ first. These are used to indicate a method name in the method look up function (in definition 4) and to describe the type of a method call (figures 16, 17, 21, 22, 31, 32) as well as other restrictions (*c.f.* figure 10), which are imposed on methods (see section 3.3.7), as given in chapters 8.2 and 9 in [8]. We also introduce the corresponding versions for constructors.

Definition 2 *For a method header* `methH` *such that*

$$\mathtt{methH} \quad = \quad \mathtt{mMod}\ \mathtt{T}\ \mathtt{m}(\mathtt{T}_1\ \mathtt{p}_1, \ldots, \mathtt{T}_n\ \mathtt{p}_n)\ \mathbf{throws}\ \mathtt{E}_1, \ldots, \mathtt{E}_s$$

we define the following functions:

$$
\begin{array}{ll}
MethId(\mathtt{methH}) & = \mathtt{m} \\
ArgT(\mathtt{methH}) & = \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \\
RetT(\mathtt{methH}) & = \mathtt{T} \\
ExcT(\mathtt{methH}) & = \{\mathtt{E}_1, \ldots, \mathtt{E}_s\} \\
Mod(\mathtt{methH}) & = \mathtt{mMod}
\end{array}
$$

Definition 3 *For a constructor header* `constrH` *such that*

$$\mathtt{constrH} \quad = \quad \mathtt{cMod}\ \mathtt{C}(\mathtt{T}_1\ \mathtt{p}_1, \ldots, \mathtt{T}_n\ \mathtt{p}_n)\ \mathbf{throws}\ \mathtt{E}_1, \ldots, \mathtt{E}_s$$

we define the following functions:

$$
\begin{aligned}
&\mathit{ConstrId}(\mathtt{constrH}) && = \mathtt{C}\\
&\mathit{ArgT}(\mathtt{constrH}) && = \mathtt{T}_1 \times \ldots \times \mathtt{T}_n\\
&\mathit{ExcT}(\mathtt{constrH}) && = \{\mathtt{E}_1, \ldots, \mathtt{E}_s\}\\
&\mathit{Mod}(\mathtt{constrH}) && = \mathtt{cMod}
\end{aligned}
$$

Definition 4 *For a declaration* $\mathtt{D}$ *with* $\vdash\ \mathtt{D} \diamond_u$ *and identifiers* $\mathtt{I}$, $\mathtt{C}$, $\mathtt{X}$, $\mathtt{m}$, $\mathtt{f}$ *we define the following look up functions:*

$$
\mathit{ISpec}(\mathtt{D},\mathtt{I}) = \begin{cases} \mathtt{iMod}\ \textbf{interface}\ \mathtt{I}\ \textbf{ext}\ \mathtt{I}_1,\ldots,\mathtt{I}_j\ \mathrm{f}\ \mathtt{iSpec}\mathrm{g} \\ \qquad\qquad \mathit{iff}\quad \mathtt{D} = \mathtt{D}'\ \mathtt{iMod}\ \textbf{interface}\ \mathtt{I}\ \textbf{ext}\ \mathtt{I}_1,\ldots,\mathtt{I}_j\ \mathrm{f}\ \mathtt{iSpec}\mathrm{g}\ \mathtt{D}'' \\ \mathit{Undef} \quad \mathit{otherwise} \end{cases}
$$

$$
\mathit{CSpec}(\mathtt{D},\mathtt{C}) = \begin{cases} \mathtt{cMod}\ \textbf{class}\ \mathtt{C}\ \textbf{ext}\ \mathtt{C}'\ \textbf{impl}\ \mathtt{I}_1,\ldots,\mathtt{I}_j\ \mathrm{f}\ \mathtt{cSpec}\mathrm{g} \\ \qquad\qquad \mathit{iff}\quad \mathtt{D} = \mathtt{D}'\ \mathtt{cMod}\ \textbf{class}\ \mathtt{C}\ \textbf{ext}\ \mathtt{C}'\ \ \textbf{impl}\ \ \mathtt{I}_1,\ldots,\mathtt{I}_j\ \mathrm{f}\ \mathtt{cSpec}\mathrm{g}\ \mathtt{D}'' \\ \mathit{Undef} \quad \mathit{otherwise} \end{cases}
$$

$$
\mathit{SupCSpec}(\mathtt{D},\mathtt{C}) = \begin{cases} \mathtt{C}' & \mathit{iff}\quad \mathit{CSpec}(\mathtt{D},\mathtt{C}) = \mathtt{cMod}\ \textbf{class}\ \mathtt{C}\ \textbf{ext}\ \mathtt{C}'\ldots\mathrm{f}\ldots\mathrm{g} \\ \mathit{Undef} & \mathit{otherwise} \end{cases}
$$

$$
\mathit{Interfs}(\mathtt{D},\mathtt{X}) = \begin{cases} \{\mathtt{I}_1, \ldots, \mathtt{I}_j\} & \mathit{iff}\quad \mathit{CSpec}(\mathtt{D},\mathtt{X}) = \mathtt{cMod}\ \textbf{class}\ \mathtt{X}\ldots\ \ \textbf{impl}\ \ \mathtt{I}_1,\ldots,\mathtt{I}_j\ \mathrm{f}\ldots\mathrm{g} \\ & \mathit{or}\quad \mathit{ISpec}(\mathtt{D},\mathtt{X}) = \mathtt{iMod}\ \textbf{interface}\ \mathtt{X}\ \textbf{ext}\ \mathtt{I}_1,\ldots,\mathtt{I}_j\ \mathrm{f}\ldots\mathrm{g} \\ \emptyset & \mathit{otherwise} \end{cases}
$$

$$
\mathit{ConstrSpec}(\mathtt{D},\mathtt{C}) = \begin{cases} \{\ \mathtt{constrH}\ \mid\ \mathit{CSpec}(\mathtt{D},\mathtt{C}) = \mathtt{cMod}\ \textbf{class}\ \mathtt{C}\ \ldots\mathrm{f}\ldots\mathtt{constrH}\ldots\mathrm{g}\quad \mathit{and} \\ \quad \mathit{ConstrId}(\mathtt{constrH}) = \mathtt{C}\ \}\quad \mathit{iff}\quad \mathit{CSpec}(\mathtt{D},\mathtt{C}) \neq \mathit{Undef} \\ \emptyset \quad \mathit{otherwise} \end{cases}
$$

$$
\mathit{MSpec}(\mathtt{D},\mathtt{X},\mathtt{m}) = \begin{cases} \{\ \mathtt{methH}\ \mid\ \mathit{ISpec}(\mathtt{D},\mathtt{X}) = \mathtt{iMod}\ \textbf{interface}\ \mathtt{X}\ \ldots\mathrm{f}\ldots\mathtt{methH}\ldots\mathrm{g}\quad \mathit{and} \\ \quad \mathit{MethId}(\mathtt{methH}) = \mathtt{m}\ \}\quad \mathit{iff}\quad \mathit{ISpec}(\mathtt{D},\mathtt{I}) \neq \mathit{Undef} \\ \\ \{\ \mathtt{methH}\ \mid\ \mathit{CSpec}(\mathtt{D},\mathtt{X}) = \mathtt{cMod}\ \textbf{class}\ \mathtt{X}\ \ldots\mathrm{f}\ldots\mathtt{methH}\ldots\mathrm{g}\quad \mathit{and} \\ \quad \mathit{MethId}(\mathtt{methH}) = \mathtt{m}\ \}\quad \mathit{iff}\quad \mathit{CSpec}(\mathtt{D},\mathtt{I}) \neq \mathit{Undef} \\ \emptyset \quad \mathit{otherwise} \end{cases}
$$

$$
\mathit{FSpec}(\mathtt{D},\mathtt{X},\mathtt{f}) = \begin{cases} \mathtt{fMod\ T\ f} & \mathit{iff}\quad \mathit{ISpec}(\mathtt{D},\mathtt{X}) = \mathtt{iMod}\ \textbf{interface}\ \mathtt{X}\ldots\mathrm{f}\ldots\ \mathtt{fMod\ T\ f;}\ \ldots\mathrm{g} \\ & \mathit{or}\quad \mathit{CSpec}(\mathtt{D},\mathtt{X}) = \mathtt{cMod}\ \textbf{class}\ \mathtt{X}\ \ldots\mathrm{f}\ldots\ \mathtt{fMod\ T\ f;}\ \ldots\mathrm{g} \\ \mathit{Undef} & \mathit{otherwise} \end{cases}
$$

3.3.5 Acyclic declarations, look up, widening

The subclass $\sqsubseteq$, implements $:_{imp}$ and subinterface $\leq$ relationships deduced from a declaration $\mathtt{D}$ are defined by the inference rules in figure 7.

Each interface is its own subinterface and the assertion $\mathtt{D} \vdash \mathtt{I} \leq \mathtt{I}$ indicates that $\mathtt{I}$ is specified as an interface in $\mathtt{D}$. The superinterface of an interface is indicated in its declaration. The subinterface relationship is transitive.

Each class in $\mathtt{D}$ is its own subclass and the assertion $\mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}$ indicates that $\mathtt{C}$ is specified as a class in $\mathtt{D}$. The direct superclass of a class is indicated in its declaration. `Object` is a predefined class. The assertion $\mathtt{D} \vdash \mathtt{C} :_{imp} \mathtt{I}$ indicates that the class $\mathtt{C}$ is specified in $\mathtt{D}$ as providing an implementation for the interface $\mathtt{I}$. The subclass relationship is transitive.

A declaration $\mathtt{D}$ is acyclic, *i.e.* $\vdash\ \mathtt{D} \diamond_a$, if the subclass and subinterface relationships are acyclic.

Definition 5 $\mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}' \sqsubseteq \mathtt{C}''$ *iff* $\mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}'$ *and* $\mathtt{D} \vdash \mathtt{C}' \sqsubseteq \mathtt{C}''$

$$\frac{\vdash \; \mathtt{D} \diamond_u \qquad \mathit{ISpec}(\mathtt{D},\mathtt{I}) = \mathtt{interfMod}\;\mathbf{interface}\;\mathtt{I}\;\mathbf{ext}\;\ldots\mathtt{I}',\ldots\mathrm{f}\ldots\mathrm{g}}{\mathtt{D} \vdash \mathtt{I} \leq \mathtt{I}, \quad \mathtt{D} \vdash \mathtt{I} \leq \mathtt{I}'} \qquad \frac{\mathtt{D} \vdash \mathtt{I} \leq \mathtt{I}' \qquad \mathtt{D} \vdash \mathtt{I}' \leq \mathtt{I}''}{\mathtt{D} \vdash \mathtt{I} \leq \mathtt{I}''}$$

$$\frac{\vdash \; \mathtt{D} \diamond_u \qquad \mathit{CSpec}(\mathtt{D},\mathtt{C}) = \mathtt{classMod}\;\mathbf{class}\;\mathtt{C}\;\mathbf{ext}\;\mathtt{C}'\;\mathbf{impl}\;\ldots\mathtt{I},\ldots\mathrm{f}\ldots\mathrm{g}}{\mathtt{D} \vdash \mathtt{nil} \sqsubseteq \mathtt{C}, \quad \mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}, \quad \mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}', \quad \mathtt{D} \vdash \mathtt{C} :_{imp} \mathtt{I}}$$

$$\frac{}{\mathtt{D} \vdash \mathtt{Object} \sqsubseteq \mathtt{Object}} \qquad \frac{\mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}'}{\mathtt{D} \vdash \mathtt{C[]} \sqsubseteq \mathtt{C'[]}} \qquad \frac{\mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}' \qquad \mathtt{D} \vdash \mathtt{C}' \sqsubseteq \mathtt{C}''}{\mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}''}$$

$$\frac{\begin{array}{lll}\mathtt{D} \vdash \mathtt{I} \leq \mathtt{I}',\; \mathtt{D} \vdash \mathtt{I}' \leq \mathtt{I} & \Longrightarrow & \mathtt{I} = \mathtt{I}' \\ \mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}',\; \mathtt{D} \vdash \mathtt{C}' \sqsubseteq \mathtt{C} & \Longrightarrow & \mathtt{C} = \mathtt{C}'\end{array}}{\vdash \; \mathtt{D} \diamond_a}$$

Figure 7: Subclasses, subinterfaces and acyclic declarations

For an acyclic declarations the following functions extract components of a class or interface :

- $\mathit{Fields}(\mathtt{D},\mathtt{X},\mathtt{f})$ indicates all field definitions for `f`, (namely, the class or interface containing the field definition, type and name of the field) which were defined in `X` or in a superclass of or an interface extended or implemented `X` and possibly hidden by `X` or another superclass.
- $\mathit{Meths}(\mathtt{D},\mathtt{X},\mathtt{m})$ indicates all method definitions for a method `m` in `X` or inherited from one of its superclasses or superinterfaces and not hidden by any of those;

Note, that shadowed variables are treated differently from overridden methods. Namely, shadowed variables are part of the set *Fields*, whereas overridden methods are not part of the set *Meths*. The reason for the difference is that shadowed variables need to be stored in the objects of subclasses (*e.g.* a `Dog` object contains an `isHealthy` field inherited from the class `Animal`, even though this field is shadowed in `Dog`), whereas overridden methods are never called by objects of the subclasses (*e.g.* for `Cat` objects the only `move()` method with no argument is that from `Cat`, whereas that defined in `Animal` is of no interest to `Cat` objects). The exception to the case is for **static** fields. These are shared in the space allocated to classes and interfaces directly. So these behave differently from overshadowed instance fields.

Definition 6 *For a declaration* `D` *satisfying* $\vdash \; \mathtt{D} \diamond_a$ *and identifiers* `X`*,* `C`*,* `m`*,* `f` *we define component look up as follows:*

$$\mathit{Fields}(\mathtt{D},\mathtt{Object},\mathtt{f}) = \emptyset$$

$$\mathit{Fields}(\mathtt{D},\mathtt{I},\mathtt{f}) = \begin{cases} \{(\mathtt{fMod\ I\ T\ f})\} \cup \mathit{Fields}(\mathtt{D},\mathtt{I}',\mathtt{f}) \;\; \mathit{for}\ \forall \mathtt{I}' \in \mathit{Interfs}(\mathtt{D},\mathtt{I}) \\ \qquad\qquad \mathit{iff} \quad \mathit{FSpec}(\mathtt{D},\mathtt{I},\mathtt{f}) = \mathtt{fMod\ T\ f} \\ \mathit{Fields}(\mathtt{D},\mathtt{I}',\mathtt{f}) \;\; \mathit{for}\ \forall \mathtt{I}' \in \mathit{Interfs}(\mathtt{D},\mathtt{I}) \\ \qquad\qquad \mathit{iff} \quad \mathit{FSpec}(\mathtt{D},\mathtt{I},\mathtt{f}) = \mathit{Undef} \end{cases}$$

$$\mathit{Fields}(\mathtt{D},\mathtt{C},\mathtt{f}) = \begin{cases} \{(\mathtt{fMod\ C\ T\ f})\} \cup \mathit{Fields}(\mathtt{D},\mathit{SupCSpec}(\mathtt{D},\mathtt{C}),\mathtt{f}) \cup \mathit{Fields}(\mathtt{D},\mathtt{I},\mathtt{f}) \\ \qquad\qquad \mathit{for}\ \forall \mathtt{I} \in \mathit{Interfs}(\mathtt{D},\mathtt{C}) \quad \mathit{iff} \quad \mathit{FSpec}(\mathtt{D},\mathtt{C},\mathtt{f}) = \mathtt{fMod\ T\ f} \\ \mathit{Fields}(\mathtt{D},\mathit{SupCSpec}(\mathtt{D},\mathtt{C}),\mathtt{f}) \cup \mathit{Fields}(\mathtt{D},\mathtt{I},\mathtt{f}) \\ \qquad\qquad \mathit{for}\ \forall \mathtt{I} \in \mathit{Interfs}(\mathtt{D},\mathtt{C}) \quad \mathit{iff} \quad \mathit{FSpec}(\mathtt{D},\mathtt{C},\mathtt{f}) = \mathit{Undef} \end{cases}$$

$$
\begin{aligned}
&\mathit{Meths}(\mathtt{D},\mathtt{Object},\mathtt{m}) &&= \emptyset \\
&\mathit{Meths}(\mathtt{D},\mathtt{X},\mathtt{m}) &&= \begin{cases}
\mathit{MSpec}(\mathtt{D},\mathtt{X},\mathtt{m}) \cup \{\ \mathtt{methH}' \mid \mathtt{methH}' \in \mathit{Meths}(\mathtt{D}, \mathit{SupCSpec}(\mathtt{D},\mathtt{X}),\mathtt{m}) \quad \mathit{and} \\
\quad \forall \mathtt{methH} \in \mathit{MSpec}(\mathtt{D},\mathtt{X},\mathtt{m}) \quad \mathit{ArgT}(\mathtt{methH}') \neq \mathit{ArgT}(\mathtt{methH})\ \} \\
\hfill \mathit{iff} \quad \mathtt{D} \vdash \mathtt{X} \sqsubseteq \mathtt{X} \\
\mathit{MSpec}(\mathtt{D},\mathtt{X},\mathtt{m}) \cup \{\ \mathtt{methH}' \mid \exists \mathtt{I}' \in \mathit{Interfs}(\mathtt{D},\mathtt{X}) \quad \mathtt{methH}' \in \mathit{Meths}(\mathtt{D},\mathtt{I}',\mathtt{m}) \\
\quad \mathit{and} \quad \forall \mathtt{methH} \in \mathit{MSpec}(\mathtt{D},\mathtt{X},\mathtt{m}) \quad \mathit{ArgT}(\mathtt{methH}') \neq \mathit{ArgT}(\mathtt{methH})\ \} \\
\hfill \mathit{iff} \quad \mathtt{D} \vdash \mathtt{X} \leq \mathtt{X}
\end{cases}
\end{aligned}
$$

The sets $\mathit{Fields}(\mathtt{D},\mathtt{Object},\mathtt{v})$ and $\mathit{Meths}(\mathtt{D},\mathtt{Object},\mathtt{m})$ should contain the standard entities described in chapter 20.1 of [8] . We defined them as empty sets for simplicity.

For the philosophers example $\mathtt{P_{Animal}}$ the above functions are:

$\mathit{Fields}(\mathtt{D}_{\mathtt{P}_{\mathtt{Animal/s}}}, \mathtt{Animal}, \mathtt{numLives})$
$= \emptyset$

$\mathit{Fields}(\mathtt{D}_{\mathtt{P}_{\mathtt{Animal/s}}}, \mathtt{Cat}, \mathtt{numLives})$
$= \{(\textbf{public}\ \mathtt{Cat}\ \textbf{int}\ \mathtt{numLives})\}$

$\mathit{Fields}(\mathtt{D}_{\mathtt{P}_{\mathtt{Animal/s}}}, \mathtt{Carnivore}, \mathtt{diet})$
$= \{(\textbf{public static final}\ \mathtt{Carnivore\ Cat\ diet})\}$

$\mathit{Fields}(\mathtt{D}_{\mathtt{P}_{\mathtt{Animal/s}}}, \mathtt{Dog}, \mathtt{isHealthy})$
$= \{(\textbf{protected}\ \mathtt{Animal}\ \textbf{boolean}\ \mathtt{isHealthy}),\ (\textbf{public}\ \mathtt{Dog}\ \textbf{boolean}\ \mathtt{isHealthy})\}$

$\mathit{Meths}(\mathtt{D}_{\mathtt{P}_{\mathtt{Animal/s}}}, \mathtt{Animal}, \mathtt{fight})$
$= \emptyset$

$\mathit{Meths}(\mathtt{D}_{\mathtt{P}_{\mathtt{Animal/s}}}, \mathtt{Cat}, \mathtt{attack})$
$= \{\ \textbf{public void}\ \mathtt{attack(Cat\ c)}\ \textbf{throws}\ ,$
$\quad \textbf{private void}\ \mathtt{attack(Dog\ d)}\ \textbf{throws}\ \}$

$\mathit{Meths}(\mathtt{D}_{\mathtt{P}_{\mathtt{Animal/s}}}, \mathtt{Carnivore}, \mathtt{eat})$
$= \{\ \textbf{public abstract void}\ \mathtt{eat(Cat\ c)}\ \textbf{throws}\ \}$

$$\frac{}{\begin{array}{l}\mathtt{D} \vdash \textbf{int} \diamond_{VarType} \\ \mathtt{D} \vdash \textbf{char} \diamond_{VarType} \\ \mathtt{D} \vdash \textbf{bool} \diamond_{VarType}\end{array}} \qquad \frac{\mathtt{D} \vdash \mathtt{I} \leq \mathtt{I}}{\mathtt{D} \vdash \mathtt{I} \diamond_{VarType}} \qquad \frac{\mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}}{\mathtt{D} \vdash \mathtt{C} \diamond_{VarType}} \qquad \frac{\mathtt{D} \vdash \mathtt{T} \diamond_{VarType}}{\mathtt{D} \vdash \mathtt{T[]} \diamond_{VarType}}$$

$$\frac{\mathtt{D} \vdash \mathtt{T_i} \diamond_{VarType} \qquad \mathtt{i} \in \{\mathtt{1...n}\}, \mathtt{n} \geq 0}{\mathtt{D} \vdash \mathtt{T_1} \times \ldots \times \mathtt{T_n} \diamond_{ArgType}}$$

$$\frac{\mathtt{D} \vdash \mathtt{T} \diamond_{VarType} \quad \mathit{or} \quad \mathtt{T} = \textbf{void} \quad \mathit{or} \quad \mathtt{T} = \mathtt{nil}}{\mathtt{D} \vdash \mathtt{T} \diamond_{NorType}} \qquad \frac{\mathtt{D} \vdash \mathtt{E_j} \sqsubseteq \mathtt{Exception} \quad \mathtt{j} \in \{\mathtt{1...s}\}, \mathtt{s} \geq 0}{\mathtt{D} \vdash \{\mathtt{E_1}, \ldots, \mathtt{E_s}\} \diamond_{AbnType}}$$

$$\frac{\begin{array}{l}\mathtt{D} \vdash \mathtt{T} \diamond_{NorType} \\ \mathtt{D} \vdash \mathtt{ET} \diamond_{AbnType}\end{array}}{\mathtt{D} \vdash \mathtt{T} \,||\, \mathtt{ET} \diamond_{Type}} \qquad \frac{\mathtt{D} \vdash \mathtt{E} \sqsubseteq \mathtt{Exception}}{\mathtt{D} \vdash \mathtt{E\text{-}Thrn} \diamond_{Type}}$$

Figure 8: Well-formed metatypes

Figure 8 defines well-formedness for the metatypes, namely, *variable types*, *argument types*, *normal types*, *abnormal types* and *types*. The assertions $\mathtt{D} \vdash \mathtt{T} \diamond_{VarType}$, $\mathtt{D} \vdash \mathtt{AT} \diamond_{ArgType}$, $\mathtt{D} \vdash \mathtt{T} \diamond_{NorType}$, $\mathtt{D} \vdash \mathtt{ET} \diamond_{AbnType}$, $\mathtt{D} \vdash \mathtt{T} \,||\, \mathtt{ET} \diamond_{Type}$, $\mathtt{D} \vdash \mathtt{E\text{-}Thrn} \diamond_{Type}$ mean correspondingly that `T` is a *variable type*, `AT` is an *argument type*, `T` is a *normal type*, `ET` is an *abnormal type*, $\mathtt{T} \,||\, \mathtt{ET}$ is a *type*, and `E-Thrn` is a *type*.

In order to reason about *abnormal types* as sets of the related exceptions, we introduce the *more specific* relation $\subseteq^e$ and the *difference* operation $\setminus^e_{\mathtt{D}}$ in definition 7. The *difference* between two *abnormal types* is the set theoretic difference, taking into account the subclass relationship.

Definition 7 *For a declaration* `D` *with* $\mathtt{D} \vdash \mathtt{ET} \diamond_{AbnType}$ *and* $\mathtt{D} \vdash \mathtt{ET}' \diamond_{AbnType}$ *we define* $\subseteq^e$ *and* $\setminus^e_{\mathtt{D}}$ *as follows:*

- $\mathtt{ET} \subseteq^e \mathtt{ET}'$ *iff* $\forall \mathtt{E} \in \mathtt{ET}\ \exists\ \mathtt{E}' \in \mathtt{ET}' :\ \mathtt{D} \vdash \mathtt{E} \sqsubseteq \mathtt{E}'$
- $\mathtt{ET}' \setminus^e_{\mathtt{D}} \mathtt{ET} = \{\ \mathtt{E}' \mid \mathtt{E}' \in \mathtt{ET}'\quad \textit{and}\quad \forall \mathtt{E} \in \mathtt{ET}\ \neg\ \mathtt{D} \vdash \mathtt{E}' \sqsubseteq \mathtt{E}\ \}$

The *more specific* relation is reflexive, transitive and antisymmetric. It is used in order to describe the situations that the **throws** clause of a method covers all possible exceptions thrown in its body or in any overriding method (see figures 10, 12 and lemma 1), *c.f.* the *abnormal types* of overriding and overriden methods `live`:

$$\begin{array}{rcl} \mathit{ExcT}(\mathit{MSpec}(\mathtt{D_{Prob/s}}, \mathtt{Doctor}, \mathtt{live})) & \subseteq^e & \mathit{ExcT}(\mathit{MSpec}(\mathtt{D_{Prob/s}}, \mathtt{Person}, \mathtt{live})),\quad \textit{i.e.} \\ \{\mathtt{Problem}, \mathtt{Illness}\} & \subseteq^e & \{\mathtt{Exception}\} \end{array}$$

The *difference* operation is used to determine which exceptions may escape out of a **try-catch-finally** statement (see figures 17, 22, 32). So, in the context of the problems example:

$$\begin{array}{rcl} \{\mathtt{Problem}\} \setminus^e_{\mathtt{D}} \{\mathtt{Illness}\} & = & \{\mathtt{Problem}\} \\ \{\mathtt{Illness}\} \setminus^e_{\mathtt{D}} \{\mathtt{Problem}\} & = & \emptyset \\ \{\mathtt{Exception}\} \setminus^e_{\mathtt{D}} \{\mathtt{Problem}, \mathtt{Illness}\} & = & \{\mathtt{Exception}\} \end{array}$$

The *union* operation is used to describe the *abnormal types* of compound statements, assignment, array creation, access, method call, exception throwing and **try-catch-finally** statements (see figures 17, 22, 32).

Widening is the reflexive projection of the subclass, subinterface and more specific relations to the *type* domain. It is defined over *variable types*, *argument types* and *types*.

Widening is expected between the left hand and right hand side of assignment (*c.f.* figures 16, 21, 31) and also between the actual and formal parameters of a method call (*c.f.* definition 15 and figure 16). Therefore, widening is defined over *variable types* in figure 9. If a *variable type* `T` can be widened to a *variable type* `T'`, as expressed by the assertion $\mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{T}'$, then a value of a *variable type* `T` can be assigned to a variable of a *variable type* type `T'` without any run-time casting or checking.

The type `nil` can be widened to any array, class or interface.

The widening relation between variable types is naturally projected to *argument types* (see the nineth rule of figure 9) and thus allows an elegant description of applicable and more special methods as in definition 15.

A *type* is the combination of a *normal type* and an *abnormal type*; therefore widening relation is defined for *types* in the last three rules in figure 9. This relation is used, particularly, for describing the *type* of a method in well-formed programs (*c.f.* figure 12), taking into account both its *return type* and the exceptions mentioned in the **throws** clause.

3.3.6 Properties of acyclic declarations

It is straightforward to state and prove the following properties of acyclic declarations. Two *variable types* that are in the subclass relationship are classes, $\sqsubseteq$ is reflexive, transitive and antisymmetric and the subclass hierarchy forms a tree. Also, two *variable types* that are in the subinterface relationship are interfaces, and $\leq$ is transitive, reflexive and antisymmetric. Unlike

$$\frac{\mathtt{D} \vdash \mathtt{T} \diamond_{VarType}}{\mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{T}} \qquad \frac{\mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{Object}}{\mathtt{D} \vdash \mathtt{nil} \leq_w \mathtt{T}}$$

$$\frac{\mathtt{D} \vdash \mathtt{T} \leq \mathtt{T}}{\mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{Object}} \qquad \frac{\mathtt{D} \vdash \mathtt{T} \sqsubseteq \mathtt{T}'}{\mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{T}'}$$

$$\frac{\mathtt{D} \vdash \mathtt{T} \leq \mathtt{T}'}{\mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{T}'} \qquad \frac{\mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{T}'}{\mathtt{D} \vdash \mathtt{T}[] \leq_w \mathtt{T}'[]}$$

$$\frac{\begin{array}{l}\mathtt{D} \vdash \mathtt{T} \sqsubseteq \mathtt{T}' \\ \mathtt{D} \vdash \mathtt{T}' :_{imp} \mathtt{T}'' \\ \mathtt{D} \vdash \mathtt{T}'' \leq \mathtt{T}'''\end{array}}{\mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{T}'''} \qquad \frac{\mathtt{D} \vdash \mathtt{T} \diamond_{VarType}}{\mathtt{D} \vdash \mathtt{T}[] \leq_w \mathtt{Object}}$$

$$\frac{\mathtt{D} \vdash \mathtt{T_i} \leq_w \mathtt{T'_i} \quad \mathtt{i} \in \{\mathtt{1...n}\}, \mathtt{n} \geq 1}{\mathtt{D} \vdash \mathtt{T_1} \times ... \times \mathtt{T_n} \leq_w \mathtt{T'_1} \times ... \times \mathtt{T'_n}}$$

$$\frac{\begin{array}{l}\mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{T}' \quad \text{or} \quad \mathtt{T} = \mathtt{T}' = \textbf{void} \\ \mathtt{D} \vdash \mathtt{ET} \subseteq^e \mathtt{ET}'\end{array}}{\mathtt{D} \vdash \mathtt{T} \parallel \mathtt{ET} \leq_w \mathtt{T}' \parallel \mathtt{ET}'}$$

$$\frac{\mathtt{D} \vdash \{\mathtt{E}\} \subseteq^e \mathtt{ET}}{\mathtt{D} \vdash \mathtt{E\text{-}Thrn} \leq_w \mathtt{T} \parallel \mathtt{ET}}$$

$$\frac{\mathtt{D} \vdash \mathtt{E} \sqsubseteq \mathtt{RTE}}{\mathtt{D} \vdash \mathtt{E\text{-}Thrn} \leq_w \mathtt{T} \parallel \mathtt{ET}}$$

Figure 9: Widening over *variable types*, *argument types*, *types*

$\sqsubseteq$, $\leq$ does not form a tree. Two *abnormal types* that are in specifics relationship are exceptions and $\subseteq^e$ 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 interace'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*.

If a *type* widens to another *type*, then the first *type* is either the combination of a *variable type* (or **void**) and an *abnormal type*, which relatively widen to the corresponding normal and abnormal components of the second *type*, or a *thrown type*, which exception class is one of subclasses of classes of the second *type*'s *abnormal type*.

3.3.7 Well-formed declarations

Figure 10 defines well-formedness of a Java program declaration. A declaration is well-formed if all the class and interface declarations are well-formed. Class and interface declarations are well-formed in `D`, iff `D` is acyclic, their components are well-formed, particularly:

- instance methods with the same identifier must have different argument types;
- a method overriding an inherited method must have the same return type as the overridden method;
- a method that overrides another method may be declared to throw 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 must be implemented either by a declaration in this class, or by an existing method declaration inherited from a direct superclass".
- when a method overrides another one, the overridden method must not be **final** and the accessibility must not be made more private by the overriding.

Definition 8 *For the access modifiers of constructors, and class and interface members, the mapping AccessMod (...) is defined in the following way:*

$$\begin{array}{ll} \mathit{AccessMod}(\{\textbf{public}...\}) & = \textbf{public} \\ \mathit{AccessMod}(\{\textbf{protected}...\}) & = \textbf{protected} \\ \mathit{AccessMod}(\{\textbf{private}...\}) & = \textbf{private} \end{array}$$

Definition 9 *Methods can be overridden only if the accessibility of the method isn't made more private. The following are allowed where a method with an access modifier from the left can override one with an access modifier from the right:*

$$\begin{array}{lll} \textbf{public} & \ll_{mod} & \{\textbf{public} \mid \textbf{protected} \mid \textbf{private}\} \\ \textbf{protected} & \ll_{mod} & \{\textbf{protected} \mid \textbf{private}\} \\ \textbf{private} & \ll_{mod} & \{\textbf{private}\} \end{array}$$

$$\frac{\begin{array}{l}\forall \mathtt{I}\quad \mathit{ISpec}(\mathtt{D},\mathtt{I}) \neq \mathit{Undef} \quad\Longrightarrow\quad \mathtt{D} \vdash \mathit{ISpec}(\mathtt{D},\mathtt{I})\diamond \\ \forall \mathtt{C}\quad \mathit{CSpec}(\mathtt{D},\mathtt{C}) \neq \mathit{Undef} \quad\Longrightarrow\quad \mathtt{D} \vdash \mathit{CSpec}(\mathtt{D},\mathtt{C})\diamond\end{array}}{\vdash \mathtt{D}\diamond}$$

$$\frac{\begin{array}{l}\vdash\ \mathtt{D}\diamond_a \\ \mathtt{I}' \in \mathit{Interfs}(\mathtt{D},\mathtt{I}) \quad\Longrightarrow\quad \mathtt{D}\vdash \mathtt{I}' \leq \mathtt{I}' \\ \mathtt{methH} \in \mathit{MSpec}(\mathtt{D},\mathtt{I},\mathtt{m}) \\ \quad\Longrightarrow\quad \mathtt{D}\vdash \mathit{RetT}(\mathtt{methH})\diamond_{VarType} \ \text{ or }\ \mathit{RetT}(\mathtt{methH}) = \mathbf{void}, \\ \qquad\qquad \mathtt{D}\vdash \mathit{ArgT}(\mathtt{methH})\diamond_{ArgType},\ \ \mathtt{D}\vdash \mathit{ExcT}(\mathtt{methH})\diamond_{ExcType} \\ \mathtt{methH}_1 \in \mathit{MSpec}(\mathtt{D},\mathtt{I},\mathtt{m}) \ \text{ and }\ \mathtt{methH}_2 \in \mathit{MSpec}(\mathtt{D},\mathtt{I},\mathtt{m}) \ \text{ and }\ \mathtt{methH}_1 \neq \mathtt{methH}_2 \\ \quad\Longrightarrow\quad \mathit{ArgT}(\mathtt{methH}_1) \neq \mathit{ArgT}(\mathtt{methH}_2) \\ \forall \mathtt{m}, \forall \mathtt{I}' \in \mathit{Interfs}(\mathtt{D},\mathtt{I}), \\ \quad \mathtt{methH} \in \mathit{MSpec}(\mathtt{D},\mathtt{I},\mathtt{m}),\ \mathtt{methH}' \in \mathit{Meths}(\mathtt{D},\mathtt{I}',\mathtt{m}),\ \mathit{ArgT}(\mathtt{methH}) = \mathit{ArgT}(\mathtt{methH}') \\ \quad\Longrightarrow\quad \mathit{RetT}(\mathtt{methH}) = \mathit{RetT}(\mathtt{methH}'),\ \mathit{ExcT}(\mathtt{methH}) \sqsubseteq^e \mathit{ExcT}(\mathtt{methH}')\end{array}}{\mathtt{D}\vdash \mathit{ISpec}(\mathtt{D},\mathtt{I})\diamond}$$

$$\frac{\begin{array}{l}\vdash\ \mathtt{D}\diamond_a \\ \mathtt{C}' = \mathit{SupCSpec}(\mathtt{D},\mathtt{C}) \quad\Longrightarrow\quad \mathtt{D}\vdash \mathtt{C}' \sqsubseteq \mathtt{C}' \\ \mathtt{I} \in \mathit{Interfs}(\mathtt{D},\mathtt{C}) \quad\Longrightarrow\quad \mathtt{D}\vdash \mathtt{I} \leq \mathtt{I} \\ \mathtt{T\ f} = \mathit{FSpec}(\mathtt{D},\mathtt{C},\mathtt{f}) \quad\Longrightarrow\quad \mathtt{D}\vdash \mathtt{T}\diamond_{VarType} \\ \mathtt{constrH} \in \mathit{ConstrSpec}(\mathtt{D},\mathtt{C}) \ \ \mathit{wtih}\ \ \mathtt{constrH} = \mathtt{C}(\mathtt{T}_1\ \mathtt{p}_1,\ldots,\mathtt{T}_n\ \mathtt{p}_n)\ \mathbf{throws}\ \mathtt{E}_1,\ldots,\mathtt{E}_s; \\ \quad\Longrightarrow\quad \mathtt{D}\vdash \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \diamond_{ArgType}, \\ \qquad\qquad \mathtt{D}\vdash \{\mathtt{E}_1,\ldots,\mathtt{E}_s\}\diamond_{AbnType} \\ \mathtt{constrH}_1 \in \mathit{ConstrSpec}(\mathtt{D},\mathtt{C}) \ \text{ and }\ \mathtt{constrH}_2 \in \mathit{ConstrSpec}(\mathtt{D},\mathtt{C}) \\ \ \ \text{and }\ \mathtt{constrH}_1 \neq \mathtt{constrH}_2 \\ \quad\Longrightarrow\quad \mathit{ArgT}(\mathtt{constrH}_1) \neq \mathit{ArgT}(\mathtt{constrH}_2) \\ \mathtt{methH} \in \mathit{MSpec}(\mathtt{D},\mathtt{C},\mathtt{m}) \ \ \mathit{with}\ \ \mathtt{methH} = \mathtt{T\ m}(\mathtt{T}_1\ \mathtt{p}_1,\ldots,\mathtt{T}_n\ \mathtt{p}_n)\ \mathbf{throws}\ \mathtt{E}_1,\ldots,\mathtt{E}_s;\mathrm{f}\,\mathtt{mBody}\mathrm{g} \\ \quad\Longrightarrow\quad \mathtt{D}\vdash \mathtt{T}\diamond_{VarType} \ \text{ or }\ \mathtt{T} = \mathbf{void}, \\ \qquad\qquad \mathtt{D}\vdash \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \diamond_{ArgType}, \\ \qquad\qquad \mathtt{D}\vdash \{\mathtt{E}_1,\ldots,\mathtt{E}_s\}\diamond_{AbnType} \\ \mathtt{methH}_1 \in \mathit{MSpec}(\mathtt{D},\mathtt{C},\mathtt{m}) \ \text{ and }\ \mathtt{methH}_2 \in \mathit{MSpec}(\mathtt{D},\mathtt{C},\mathtt{m}) \ \text{ and }\ \mathtt{methH}_1 \neq \mathtt{methH}_2 \\ \quad\Longrightarrow\quad \mathit{ArgT}(\mathtt{methH}_1) \neq \mathit{ArgT}(\mathtt{methH}_2) \\ \forall \mathtt{m}\ \ \mathtt{methH} \in \mathit{MSpec}(\mathtt{D},\mathtt{C},\mathtt{m}),\ \mathtt{methH}' \in \mathit{Meths}(\mathtt{D},\mathtt{C}',\mathtt{m}),\ \mathit{ArgT}(\mathtt{methH}) = \mathit{ArgT}(\mathtt{methH}') \\ \quad\Longrightarrow\quad \mathit{RetT}(\mathtt{methH}) = \mathit{RetT}(\mathtt{methH}'),\ \mathit{ExcT}(\mathtt{methH}) \sqsubseteq^e \mathit{ExcT}(\mathtt{methH}'), \\ \qquad\qquad \mathtt{final} \notin \mathit{Mod}(\mathtt{methH}),\ \mathit{AccessMod}(\mathit{Mod}(\mathtt{methH})) \ll_{mod} \mathit{AccessMod}(\mathit{Mod}(\mathit{methH}')) \\ \forall \mathtt{m}, \forall \mathtt{I} \in \mathit{Interfs}(\mathtt{D},\mathtt{C}),\ \mathtt{methH}_1 \in \mathit{MSpec}(\mathtt{D},\mathtt{I},\mathtt{m}) \\ \quad\Longrightarrow\quad \exists \mathtt{methH}_2 \in \mathit{Meths}(\mathtt{D},\mathtt{C},\mathtt{m}),\ \mathit{ArgT}(\mathtt{methH}_1) = \mathit{ArgT}(\mathtt{methH}_2), \\ \qquad\qquad \mathit{RetT}(\mathtt{methH}_1) = \mathit{RetT}(\mathtt{methH}_2),\ \mathit{ExcT}(\mathtt{methH}_2) \sqsubseteq^e \mathit{ExcT}(\mathtt{methH}_1)\end{array}}{\mathtt{D}\vdash \mathit{CSpec}(\mathtt{D},\mathtt{C})\diamond}$$

Figure 10: Well-formed declarations

3.3.8 Properties of well-formed declarations

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

Lemma 1 *For a declaration* $\mathtt{D}$ *with* $\vdash \mathtt{D} \diamond$ *iff* $\mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{T}'$, $\mathtt{methH}' \in \mathit{Meths}(\mathtt{D}, \mathtt{T}', \mathtt{m})$, *then* $\exists \mathtt{methH} \in \mathit{Meths}(\mathtt{D}, \mathtt{T}, \mathtt{m})$ *so that* $\mathit{ArgT}(\mathtt{methH}) = \mathit{ArgT}(\mathtt{methH}')$, $\mathit{RetT}(\mathtt{methH}) = \mathit{RetT}(\mathtt{methH}')$, $\mathit{ExcT}(\mathtt{methH}) \subseteq^e \mathit{ExcT}(\mathtt{methH}')$.

3.3.9 Environments

$$\mathit{Environment} \quad ::= \quad \epsilon \;\mid\; \mathit{VarType}\ \mathtt{VarId};\ \mathit{Environment}$$

$$\frac{\begin{array}{l} \vdash \mathtt{D} \diamond \\ \forall \mathtt{id}: \quad \Gamma = \Gamma_1;\ \mathtt{T}_1\ \mathtt{id};\ \Gamma_3, \quad \Gamma = \Gamma_2;\ \mathtt{T}_2\ \mathtt{id};\ \Gamma_4 \\ \qquad \Longrightarrow \qquad \Gamma_1 = \Gamma_2,\ \Gamma_3 = \Gamma_4,\ \mathtt{D} \vdash \mathtt{T}_1 \diamond_{VarType} \end{array}}{\mathtt{D} \vdash \Gamma \diamond}$$

$$\frac{\begin{array}{l} \mathtt{D} \vdash \Gamma \diamond \\ \Gamma = \ \Gamma_1;\ \mathtt{T}\ \mathtt{id};\ \Gamma_3 \end{array}}{\Gamma(\mathtt{id}) = \mathtt{T}\ \mathtt{id}} \qquad \frac{\begin{array}{l} \mathtt{D} \vdash \Gamma \diamond \\ \Gamma \neq \ \Gamma_1;\ \mathtt{T}\ \mathtt{id};\ \Gamma_3,\ \text{for all}\ \Gamma_1, \Gamma_3 \end{array}}{\Gamma(\mathtt{id}) = \mathit{Undef}}$$

Figure 11: Environments

Any variable, used in a program, should be declared to have a *variable type*. Variable declarations introduce variables of given types. In order to describe well-formed variable declarations we introduce the concept of *environment* of computation determined by a declaration $\mathtt{D}$.

Method parameters should be declared to have a *variable type*. In order to describe well-formed parameter declarations we introduce the concept of *environment* of computation determined by a declaration $\mathtt{D}$.

An environment, defined in figure 11, consists of variable declarations and is usually denoted by Γ, Γ', Γ_1, *etc.* The environment is well-formed, *i.e.* $\mathtt{D} \vdash \Gamma \diamond$, if the variable declarations are unambiguous and the types are types in the sense of $\mathtt{D}$. In well-formed environments look up of variables – $\Gamma(\mathtt{id})$ – gives their types.

3.3.10 Typing of Java terms

Thus, typing of a Java term $\mathtt{t}$ takes place in the context of declaration $\mathtt{D}$ and an environment Γ:

$$\text{Declaration} \times \text{Environment} \vdash \ \text{term} : \text{Type}$$

and has the form:

$$\mathtt{D}, \Gamma \vdash \ \mathtt{t} : \tau$$

The term $\mathtt{t}$ and *type* τ are considered for each language Java$_s$, Java$_{se}$, Java$_r$. The inference of *normal type* and *abnormal type*, the *type* consists of, is indicated by the index $/n$ index $/e$ correspondingly. Thus, in order to distinguish the type inference rules between the languages Java$_s$, Java$_{se}$, Java$_r$ and the metatypes we introduce the following 9 kinds of inferences:

- $\vdash_{s/n}$ inference of a *normal type* of a term in Java$_s$
- $\vdash_{s/e}$ inference of an *abnormal type* of a term in Java$_s$
- $\vdash_{s}$ inference of a *type* of a term in Java$_s$
- $\vdash_{se/n}$ inference of a *normal type* of a term in Java$_{se}$

- $\vdash_{se/e}$ inference of an *abnormal type* of a term in Java$_{se}$
- $\vdash_{se}$ inference of a *type* of a term in Java$_{se}$
- $\vdash_{r/n}$ inference of a *normal type* of a term in Java$_r$
- $\vdash_{r/e}$ inference of an *abnormal type* of a term in Java$_r$
- $\vdash_{r}$ inference of a *type* of a term in Java$_r$.

3.3.11 Well-formed programs

In order to describe well-formedness of programs and also the operational semantics (in section 3.7.3) we need additional look-up functions concerned with method bodies. But first we introduce auxiliary functions to extract the method components, namely, a method identifier, an *argument type* and a method body (similar to those introduced for a method declaration in definition 11). We also introduce corresponding versions to extract the constructor components as in definition 10.

Definition 10 *For a constructor* `constr` *such that*

$$\texttt{constr} = \texttt{constrMod C(T}_1\ \texttt{p}_1\texttt{,...,T}_n\ \texttt{p}_n\texttt{)}\ \textbf{throws}\ \texttt{E}_1\texttt{,...,E}_s\ \{\texttt{cBody}\}$$

we define the following functions:

$$\begin{aligned}
\mathit{ConstrId}(\texttt{constr}) &= \texttt{C} \\
\mathit{ArgT}(\texttt{constr}) &= \texttt{T}_1 \times \ldots \times \texttt{T}_n \\
\mathit{ConstrBody}(\texttt{constr}) &= \texttt{cBody}
\end{aligned}$$

Definition 11 *For a method* `meth` *such that*

$$\texttt{meth} = \texttt{methMod T m(T}_1\ \texttt{p}_1\texttt{,...,T}_n\ \texttt{p}_n\texttt{)}\ \textbf{throws}\ \texttt{E}_1\texttt{,...,E}_s\ \{\texttt{mBody}\}$$

we define the following functions:

$$\begin{aligned}
\mathit{MethId}(\texttt{meth}) &= \texttt{m} \\
\mathit{ArgT}(\texttt{meth}) &= \texttt{T}_1 \times \ldots \times \texttt{T}_n \\
\mathit{MBody}(\texttt{meth}) &= \texttt{mBody}
\end{aligned}$$

Definition 12 *For a program* `P`*, its declaration* `D` *with* $\vdash\ \texttt{D}\ \diamond_u$ *and identifiers* `C`*,* `m` *we define the following look up functions:*

$$
\mathit{IDef}(\mathtt{P},\mathtt{I}) = \begin{cases} \mathtt{iMod}\ \textbf{interface}\ \mathtt{I}\ \textbf{ext}\ \mathtt{I_1,\ldots,I_j}\,\mathrm{f}\ \mathtt{iBody}\mathrm{g} \\ \qquad\qquad \mathit{iff}\quad \mathtt{P} = \mathtt{P'}\ \mathtt{iMod}\ \textbf{interface}\ \mathtt{I}\ \textbf{ext}\ \mathtt{I_1,\ldots,I_j}\,\mathrm{f}\ \mathtt{iBody}\mathrm{g}\ \mathtt{P''} \\ \mathit{Undef} \quad \mathit{otherwise} \end{cases}
$$

$$
\mathit{CDef}(\mathtt{P},\mathtt{C}) = \begin{cases} \mathtt{cMod}\ \textbf{class}\ \mathtt{C}\ \textbf{ext}\ \mathtt{C'}\ \textbf{impl}\ \mathtt{I_1,\ldots,I_j}\,\mathrm{f}\ \mathtt{cBody}\mathrm{g} \\ \qquad\qquad \mathit{iff}\quad \mathtt{P} = \mathtt{P'}\ \mathtt{cMod}\ \textbf{class}\ \mathtt{C}\ \textbf{ext}\ \mathtt{C'}\ \ \textbf{impl}\ \ \mathtt{I_1,\ldots,I_j}\,\mathrm{f}\ \mathtt{cBody}\mathrm{g}\ \mathtt{P''} \\ \mathit{Undef} \quad \mathit{otherwise} \end{cases}
$$

$$
\mathit{SuperC}(\mathtt{P},\mathtt{C}) = \begin{cases} \mathtt{C'} & \mathit{iff}\quad \mathit{CDef}(\mathtt{P},\mathtt{C}) = \mathtt{cMod}\ \textbf{class}\ \mathtt{C}\ \textbf{ext}\ \mathtt{C'}\ldots\mathrm{f}\ldots\mathrm{g} \\ \mathit{Undef} & \mathit{otherwise} \end{cases}
$$

$$
\mathit{ConstrDef}(\mathtt{P},\mathtt{C}) = \begin{cases} \{\ \mathtt{constr}\ \mid\ \mathit{CDef}(\mathtt{P},\mathtt{C}) = \mathtt{cMod}\ \textbf{class}\ \mathtt{C}\ \ldots\mathrm{f}\ldots\mathtt{constr}\ldots\mathrm{g}\ \} \\ \emptyset \quad \mathit{otherwise} \end{cases}
$$

$$
\mathit{MDef}(\mathtt{P},\mathtt{X},\mathtt{m}) = \begin{cases} \{\ \mathtt{meth}\ \mid\ \mathit{CDef}(\mathtt{P},\mathtt{X}) = \mathtt{cMod}\ \textbf{class}\ \mathtt{X}\ \ldots\mathrm{f}\ldots\mathtt{meth}\ldots\mathrm{g}\ \} \\ \{\ \mathtt{meth}\ \mid\ \mathit{IDef}(\mathtt{P},\mathtt{X}) = \mathtt{iMod}\ \textbf{interface}\ \mathtt{X}\ \ldots\mathrm{f}\ldots\mathtt{meth}\ldots\mathrm{g}\ \} \\ \emptyset \quad \mathit{otherwise} \end{cases}
$$

$$
\begin{aligned}
\mathit{Meths}(\mathtt{P},\mathtt{C},\mathtt{m}) &= \mathit{MDef}(\mathtt{P},\mathtt{C},\mathtt{m}) \cup \{\ \mathtt{meth'}\ \mid \mathtt{meth'} \in \mathit{Meths}(\mathtt{P}, \mathit{SuperC}(\mathtt{P},\mathtt{C}),\mathtt{m})\quad \mathit{and} \\
&\quad \forall \mathtt{meth} \in \mathit{MDef}(\mathtt{P},\mathtt{C},\mathtt{m})\quad \mathit{ArgT}(\mathtt{meth'}) \neq \mathit{ArgT}(\mathtt{meth})\ \}\quad \mathit{iff}\quad \mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}
\end{aligned}
$$

Figure 12 defines well-formedness of a Java program. Programs are well-formed if their declarations are well-formed and method bodies have a *type* compatible with the *type* declared in their headers.

$$
\begin{array}{l}
\mathcal{D}(\mathtt{P}) = \mathtt{D} \\
\vdash \mathtt{D_{FUIO}}\ \mathtt{D} \diamond \\
\forall \mathtt{I} \quad \mathit{IDef}(\mathtt{P},\mathtt{I}) \neq \mathit{Undef} \quad \Longrightarrow \quad \mathtt{D_{FUIO}}\ \mathtt{D} \vdash \mathit{IDef}(\mathtt{P},\mathtt{I}) \diamond \\
\forall \mathtt{C} \quad \mathit{CDef}(\mathtt{P},\mathtt{C}) \neq \mathit{Undef} \quad \Longrightarrow \quad \mathtt{D_{FUIO}}\ \mathtt{D} \vdash \mathit{CDef}(\mathtt{P},\mathtt{C}) \diamond \\
\hline
\vdash \mathtt{P} \diamond
\end{array}
$$

$$
\begin{array}{l}
\mathcal{D}(\mathtt{P}) = \mathtt{D} \\
\vdash \mathtt{D} \diamond \\
\mathtt{meth} \in \mathit{MDef}(\mathtt{P},\mathtt{I},\mathtt{m}) \quad \mathit{with} \quad \mathtt{meth} = \mathtt{mMod\ T\ m(T}_1\ \mathtt{p}_1, \ldots, \mathtt{T}_n\ \mathtt{p}_n\mathtt{)}\ \textbf{throws}\ \mathtt{E_1, \ldots, E_s;} \\
\quad \Longrightarrow \quad \mathtt{p_i} \neq \mathtt{this} \quad \mathtt{i} \in \{1 \ldots \mathtt{n}\} \\
\qquad\qquad \mathtt{D} \vdash \mathtt{T_1\ p_1; \ldots; T}_n\ \mathtt{p}_n \diamond \\
\hline
\mathtt{D} \vdash \mathit{IDef}(\mathtt{P},\mathtt{I}) \diamond
\end{array}
$$

$$
\begin{array}{l}
\mathcal{D}(\mathtt{P}) = \mathtt{D} \\
\vdash \mathtt{D} \diamond \\
\mathtt{constr} \in \mathit{ConstrDef}(\mathtt{P},\mathtt{C}) \quad \mathit{with} \quad \mathtt{constr} = \mathtt{constrMod\ C(T}_1\ \mathtt{p}_1, \ldots, \mathtt{T}_n\ \mathtt{p}_n\mathtt{)} \\
\qquad\qquad \textbf{throws}\ \mathtt{E_1, \ldots, E_s f\, constrBodyg} \\
\quad \Longrightarrow \quad \mathtt{p_i} \neq \mathtt{this} \quad \mathtt{i} \in \{1 \ldots \mathtt{n}\} \\
\qquad\qquad \mathtt{D} \vdash \mathtt{T_1\ p_1; \ldots; T}_n\ \mathtt{p}_n \diamond \\
\qquad\qquad \exists \mathtt{T}', \exists \mathtt{ET}' \ \ \mathtt{D,\ T_1\ p_1; \ldots; T_n\ p_n} \vdash_s \mathtt{constrBody : void} \parallel \mathtt{ET}' \ \ \text{and} \ \ \mathtt{D} \vdash \mathtt{ET}' \leq_w \{\mathtt{E_1}, \ldots, \mathtt{E_s}\} \\
\mathtt{meth} \in \mathit{MDef}(\mathtt{P},\mathtt{C},\mathtt{m}) \quad \mathit{with} \quad \mathtt{meth} = \mathtt{mMod\ T\ m(T}_1\ \mathtt{p}_1, \ldots, \mathtt{T}_n\ \mathtt{p}_n\mathtt{)}\ \textbf{throws}\ \mathtt{E_1, \ldots, E_s;} \\
\quad \Longrightarrow \quad \mathtt{p_i} \neq \mathtt{this} \quad \mathtt{i} \in \{1 \ldots \mathtt{n}\} \\
\qquad\qquad \textbf{abstract} \in \mathtt{mMod} \\
\qquad\qquad \mathtt{D} \vdash \mathtt{T_1\ p_1; \ldots; T}_n\ \mathtt{p}_n \diamond \\
\mathtt{meth} \in \mathit{MDef}(\mathtt{P},\mathtt{C},\mathtt{m}) \quad \mathit{with} \quad \mathtt{meth} = \mathtt{mMod\ T\ m(T}_1\ \mathtt{p}_1, \ldots, \mathtt{T}_n\ \mathtt{p}_n\mathtt{)} \\
\qquad\qquad \textbf{throws}\ \mathtt{E_1, \ldots, E_s f\, mBodyg} \\
\quad \Longrightarrow \quad \mathtt{p_i} \neq \mathtt{this} \quad \mathtt{i} \in \{1 \ldots \mathtt{n}\} \\
\qquad\qquad \mathtt{D} \vdash \mathtt{T_1\ p_1; \ldots; T}_n\ \mathtt{p}_n \diamond \\
\qquad\qquad \exists \mathtt{T}', \exists \mathtt{ET}' \ \ \mathtt{D,\ T_1\ p_1; \ldots; T_n\ p_n} \vdash_s \mathtt{mBody : T}' \parallel \mathtt{ET}' \ \ \text{and} \ \ \mathtt{D} \vdash \mathtt{T}' \parallel \mathtt{ET}' \leq_w \mathtt{T} \parallel \{\mathtt{E_1}, \ldots, \mathtt{E_s}\} \\
\hline
\mathtt{D} \vdash \mathit{CDef}(\mathtt{P},\mathtt{C}) \diamond
\end{array}
$$

Figure 12: Well-formed Java_s programs

3.4 Java$_s$ types

The type rules of Java$_s$ correspond to the type checking phase of a Java compiler and have one of the following forms:
• $\mathtt{D}, \Gamma \vdash_{s/n} \mathtt{t} : \mathtt{T}$; it means that *normal execution* of the term t may lead to a value of *normal type* T;
• $\mathtt{D}, \Gamma \vdash_{s/e} \mathtt{t} : \mathtt{ET}$; it means that *abnormal execution* of the term t may lead to a value of one of subclasses of classes from *abnormal type* ET;
• $\mathtt{D}, \Gamma \vdash_{s} \mathtt{t} : \mathtt{T} \,||\, \mathtt{ET}$; it means that execution of the term t leads to a value of the *normal type* T with *abnormal type* ET.
Note the subscripts under the $\vdash$ which is used in order to distinguish various type systems and indicate the *normal type* and *abnormal type* inference in the Java$_s$ language.

3.4.1 Normal types of Java$_s$

According to the rules on the first line of figure 16, boolean literals have boolean type, character literals have character type, integer literals have integer type, identifiers have the type as declared in the environment. The `null` literal has `nil` type.

According to the rule on the second line, a statement sequence has the same type as its last statement. According to the first rule on the third line, an expression of type $\mathtt{T}'$ can be assigned to a variable of type T if $\mathtt{T}'$ can be widened to T and the assignment is not made to a field declared **final**.

Figure 13 shows all possible ways of determining if an assignment is attempted on a **final** field of any object, class or interface.

$$\frac{\mathtt{v} \neq \mathtt{X.f} \quad \text{or} \quad \mathtt{v} = \mathtt{X}[f]}{\mathit{NonFinalField}(\mathtt{D}, \mathtt{v})} \qquad \frac{\begin{array}{l} \mathtt{D}, \Gamma \vdash_{s/n} \mathtt{v} : \mathtt{T} \\ \mathit{FSpec}(\mathtt{D}, \mathtt{T}, \mathtt{f}) = (\mathtt{Mod\ X\ f}) \\ \mathbf{final} \notin \mathtt{Mod} \end{array}}{\mathit{NonFinalField}(\mathtt{D}, \mathtt{v.f})}$$

$$\frac{\begin{array}{l} \mathtt{D}, \Gamma \vdash_{s/n} \mathtt{v} : \mathtt{T} \\ \mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{C} \\ \mathit{FSpec}(\mathtt{D}, \mathtt{C}, \mathtt{f}) = (\mathtt{Mod\ Cf}) \\ \mathbf{final} \notin \mathtt{Mod} \end{array}}{\mathit{NonFinalField}(\mathtt{D}, \mathtt{v}[\mathtt{C}].\mathtt{f})} \qquad \frac{\begin{array}{l} \mathtt{D} \vdash \mathtt{X} \leq_w \mathtt{X} \\ \mathit{FSpec}(\mathtt{D}, \mathtt{X}, \mathtt{f}) = (\mathtt{Mod\ X\ f}) \\ \mathbf{final} \notin \mathtt{Mod} \end{array}}{\mathit{NonFinalField}(\mathtt{D}, \mathtt{X.f})}$$

Figure 13: Non-final field of class or interface

A conditional consists of two statement sequences not necessarily of the same type. The other two rules on the third line describe the type of return statements. A return statement has **void** type, or the same type as the expression it returns.

The fourth line describes the type of constructor invocation. A constructor is *applicable* if the actual parameters can be widened to the corresponding formal parameter types. The more specific applicable constructors are contained in the set *MostSpecConstr*. This expression is type-correct when the set is unary. For a non-**abstract** class C, the expression **new** $\mathtt{C}(e_1 ... e_n)$ has type C if the *most special constructor* as determined by the arguments of the expression is accessible from the point in the program where the expression lies.

Figure 14 gives the definition of a non **abstract** class.

Definition 13 *For a declaration* D *satisfying* $\vdash\ \mathtt{D} \diamond_a$, *class* C, *argument type* AT, *the* most special constructor *is defined as follows:*

- $\mathit{ApplConstrs}(\mathtt{D}, \mathtt{C}, \mathtt{AT}) = \{\ \mathtt{constrH} \ \mid\ \mathtt{constrH} \in \mathit{ConstrSpec}(\mathtt{D}, \mathtt{C})$ *and* $\mathtt{D} \vdash \mathit{ArgT}(\mathtt{constrH}) \leq_w \mathtt{AT}\ \}$

$$\begin{array}{l}\vdash\ \mathtt{D}\ \diamond_a\\ \mathit{CSpec}(\mathtt{D},\mathtt{C}) = \mathtt{cMod}\ \textbf{class}\ \mathtt{C}\ \textbf{ext}\ \mathtt{C}'...\mathrm{f}...\mathrm{g}\\ \textbf{abstract} \notin \mathtt{cMod}\\ \hline \mathit{NonAbstractClass}(\mathtt{D},\mathtt{C})\end{array}$$

Figure 14: Non-abstract class

- $\mathit{MostSpecConstr}(\mathtt{D},\mathtt{C},\mathtt{AT}) = \{\ \mathtt{constrH}\ \mid\ \mathtt{constrH} \in \mathit{ApplConstrs}(\mathtt{D},\mathtt{C},\mathtt{AT})\ \ \mathit{and}$ $\not\exists \mathtt{constrH}' \neq \mathtt{constrH}\ \ \mathit{such\ that}\ \ \mathtt{D} \vdash \mathit{ArgT}(\mathtt{constrH}') \leq_w \mathit{ArgT}(\mathtt{constrH})\ \}$

Figure 15 shows when a member of constructor of an object or class or interface X is accessible from some class (possibly the same) in the program.

$$\begin{array}{l}\vdash\ \mathtt{D}\ \diamond_a\\ \mathtt{D} \vdash \mathtt{X} \sqsubseteq \mathtt{X}\ \ \text{or}\ \ \mathtt{D} \vdash \mathtt{X} \leq \mathtt{X}\\ \textbf{private} \in \mathit{Mod}\\ \hline \mathit{Accessible}(\mathtt{D},\mathtt{X},\mathtt{X},\mathtt{Mod})\end{array} \qquad \begin{array}{l}\vdash\ \mathtt{D}\ \diamond_a\\ \mathtt{D} \vdash \mathtt{X} \sqsubseteq \mathtt{X}\ \ \text{or}\ \ \mathtt{D} \vdash \mathtt{X} \leq \mathtt{X}\\ \textbf{public} \in \mathit{Mod}\\ \hline \mathit{Accessible}(\mathtt{D},\mathtt{C},\mathtt{X},\mathtt{Mod})\end{array}$$

Figure 15: Accessibility of members of classes and interfaces

Line five deals with array creation. For a simple type T, the expression **new** $\mathtt{T}[\mathtt{e_1}]...[\mathtt{e_n}][]_1...[]_k$ is a n+k-dimensional array of elements of type T.

The sixth line describes the type of field and array access. For an array access v[e] the variable v should have an array type T[], and e should be of integer type. For a field access v.f the variable v should have a class type T, one of whose superclasses (C) should contain a field declaration for f as of type T′ and modifier(s) fMod such that the field is accessible from that point in the program. Moreover, it should be such a declaration which is *first visible*, *i.e.* f is specified in a superclass C′ and there is no other class C″ between the original class and the superclass which contains another field declaration for f. For a field access X.f, X should be a class or interface one of that has a **static** field declaration for f as of type T and modifier(s) fMod such that the field is accessible from that point in the program.

Definition 14 *For a declaration* D *satisfying* $\vdash\ \mathtt{D}\ \diamond_a$, *identifier* m, *classes or interfaces* X, Y, Z, *variable type* T *we define*

$$\begin{array}{l}\mathit{FirstVisSet}(\mathtt{D},\mathtt{X},\mathtt{f}) = \{(\mathtt{fMod\ Y\ T\ f})\ \mid\ (\mathtt{fMod\ Y\ T\ f}) \in \mathit{Fields}(\mathtt{D},\mathtt{X},\mathtt{f})\ \ \mathit{and}\\ \qquad\qquad \forall \mathtt{Z} \neq \mathtt{Y},\ \mathtt{Z} \neq \mathtt{X}\ \ \mathtt{D} \vdash \mathtt{X} \leq_w \mathtt{Z} \leq_w \mathtt{Y}\ \Longrightarrow\ \mathit{FSpec}(\mathtt{D},\mathtt{Z},\mathtt{f}) = \mathit{Undef}\}\end{array}$$

$$\mathit{FirstVis}(\mathtt{D},\mathtt{X},\mathtt{f}) = \begin{cases}(\mathtt{fMod\ Y\ T\ f}) & \mathit{iff}\quad \mathit{FirstVisSet}(\mathtt{D},\mathtt{X},\mathtt{f}) = \{(\mathtt{fMod\ Y\ T\ f})\}\\ \mathit{Undef} & \mathit{otherwise}\end{cases}$$

This definition helps to show how ambiguities in references to inherited variables can be so easily detected. Also if the field f is first visible in the class hierarchy, *i.e.* $\mathit{FirstVis}(\mathtt{D},\mathtt{T},\mathtt{f}) = (\mathtt{C\ T'\ f})$, then the field access expression has the type T′.

The first rule in the seventh line of figure 16 describes method call. 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 15 *For a declaration* D *satisfying* $\vdash\ \mathtt{D}\ \diamond_a$, *identifier* m, *variable type* T, *argument type* AT, *the* most special methods *are defined as follows:*

- $\mathit{ApplMeths}(\mathtt{D},\mathtt{m},\mathtt{T},\mathtt{AT}) = \{\ (\mathtt{T}',\ \mathtt{methH})\ \mid\ \mathtt{methH} \in \mathit{Meths}(\mathtt{D},\mathtt{T},\mathtt{m})\ \ \mathit{and}\ \ \mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{T}'\ \ \mathit{and}$ $\mathtt{D} \vdash \mathit{ArgT}(\mathtt{methH}) \leq_w \mathtt{AT}\ \}$

- $(\mathtt{T},\mathtt{methH}) \ll (\mathtt{T}',\ \mathtt{methH}')\quad \mathit{iff}\quad \mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{T}'\ \ \mathit{and}$ $\mathtt{D} \vdash \mathit{ArgT}(\mathtt{methH}) \leq_w \mathit{ArgT}(\mathtt{methH}')$

- $\mathit{MostSpec}(\mathtt{D},\mathtt{m},\mathtt{T},\mathtt{AT}) = \{\ \mathtt{methH}\ \mid\ (\mathtt{T},\ \mathtt{methH}) \in \mathit{ApplMeths}(\mathtt{D},\mathtt{m},\mathtt{T},\mathtt{AT})\ \ \mathit{and}$ $\nexists(\mathtt{T}',\ \mathtt{methH}') \neq (\mathtt{T},\ \mathtt{methH})\ \ \mathit{such\ that}\ \ (\mathtt{T}',\ \mathtt{methH}') \ll (\mathtt{T},\ \mathtt{methH})\ \}$

The more specific applicable methods are contained in the set *MostSpec*. A message expression is type-correct when this set contains exactly one method. The argument types of the signature of this pair is stored as the *method descriptor*, *c.f.* ch.15.11 in [8], and the result type of the signature is the type of the message expression.

The last two last in figure 16 describe the *normal type* for statements dealing with exception throwing and catching. A **throw** statement has the type **void**, if the expression following the **throw** indicates an exception. Similarly, the **try**-**catch**-**finally** statements have the type **void**, provided that the constituent statement lists are well-typed, and that the names of exception classes and new variables appear after each **catch**. The additional Java requirements, that no class $\mathtt{E_i}$ should appear more than once, and that no class should be followed by a subclass are omitted here.

3.4.2 Abnormal types of Java_s

Figure 17 describes the *abnormal types* for Java_s terms.

According to the rules on the first line, values of primitive types, identifiers and the `null` literal have the empty *abnormal type*. According to the rules on the remaining lines 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.

According to the rule on the fifth line, the *abnormal type* of a constructor invocation expression is the union of the *abnormal types* of its arguments and of the *abnormal type* declared in the method header.

The *abnormal type* of an array creation expression consists of the *abnormal types* of the subexpressions only and not the *abnormal type* of a constructor because array creation doesn't actually involve the invocation of any constructor.

The *exception type* of an array element access `v[e]` is the union of the *exception types* of variable `v` and expression `e`. The *exception type* of a field access `v.f` consists of the *exception type* of a variable `v`, but when the field access is by the qualified name of a static variable *e.g.* `X.f`, the *exception type* is empty.

According to the first rule on the eighth line, the *abnormal type* of a method call is the union of the *abnormal types* of its arguments and of the *abnormal type* declared in the method header.

The next two rules describe the *abnormal types* for statements dealing with exception throwing and catching. A **throw** statement contains an expression; evaluated it will create and throw an object of subclass of exception which will be thrown. The *abnormal type* of a **throw** statement 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 the **try**-**catch**-**finally** statement consists of the *abnormal type* of **try**, **catch** and **finally** blocks' statements without exception classes $\mathtt{E_1}$, ..., $\mathtt{E_n}$ actually being caught.

3.4.3 Properties of the Java_s type system

Definition 16 $\mathtt{D},\Gamma \vdash_s \mathtt{t} : \mathtt{T} \,\|\, \mathtt{ET}\quad \mathit{iff}\quad \mathtt{D},\Gamma \vdash_{s/n} \mathtt{t} : \mathtt{T}\ \ \mathit{and}\ \ \mathtt{D},\Gamma \vdash_{s/e} \mathtt{t} : \mathtt{ET}$

Definition 17 *For a Java_s term* `t`*, a program* `P`*, declaration* `D` *with* $\mathtt{D} = \mathcal{D}(\mathtt{P})$*, environment* Γ *we define:*

$$\frac{\texttt{i}\text{ is integer},\quad \texttt{c}\text{ is character},\quad \Gamma(\texttt{x}) = \texttt{T x}}{\begin{array}{l}\texttt{D},\Gamma \vdash_{s/n} \texttt{true} : \textbf{bool},\quad \texttt{D},\Gamma \vdash_{s/n} \texttt{false} : \textbf{bool},\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{i} : \textbf{int},\quad \texttt{D},\Gamma \vdash_{s/n} \texttt{c} : \textbf{char},\quad \texttt{D},\Gamma \vdash_{s/n} \texttt{x} : \texttt{T}\end{array}} \qquad \frac{}{\texttt{D},\Gamma \vdash_{s/n} \texttt{null} : \texttt{nil}}$$

$$\frac{\begin{array}{l}\texttt{D},\Gamma \vdash_{s/n} \texttt{e} : \textbf{bool}\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{stmts} : \textbf{void} \qquad \texttt{D},\Gamma \vdash_{s/n} \texttt{stmts}' : \textbf{void} \qquad \texttt{D},\Gamma \vdash_{s/n} \texttt{stmt} : \texttt{T}\end{array}}{\begin{array}{l}\texttt{D},\Gamma \vdash_{s/n} \texttt{stmts ; stmt} : \texttt{T}\\ \texttt{D},\Gamma \vdash_{s/n} \textbf{if}\ \texttt{e}\ \textbf{then}\ \texttt{stmts}\ \textbf{else}\ \texttt{stmts}' : \textbf{void}\end{array}}$$

$$\frac{\begin{array}{l}\texttt{D},\Gamma \vdash_{s/n} \texttt{v} : \texttt{T}\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{e} : \texttt{T}'\\ \texttt{D} \vdash \texttt{T}' \leq_w \texttt{T}\\ \mathit{NonFinalField}(\texttt{D},\texttt{v})\end{array}}{\texttt{D},\Gamma \vdash_{s/n} \texttt{v=e} : \textbf{void}} \qquad \frac{\texttt{D},\Gamma \vdash_{s/n} \texttt{e} : \texttt{T}}{\texttt{D},\Gamma \vdash_{s/n} \textbf{return}\ \texttt{e} : \texttt{T}} \qquad \frac{}{\texttt{D},\Gamma \vdash_{s/n} \textbf{return} : \textbf{void}}$$

$$\frac{\begin{array}{l}\texttt{D} \vdash \texttt{C} \sqsubseteq \texttt{C},\quad \mathit{NonAbstractClass}(\texttt{D},\texttt{C})\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{e}_\texttt{i} : \texttt{T}_\texttt{i} \quad \texttt{i} \in \{1...\texttt{n}\}, \texttt{n} \geq 1\\ \mathit{MostSpecConstr}(\texttt{D},\texttt{C},\texttt{T}_1 \times ... \times \texttt{T}_\texttt{n}) = \{\texttt{constrH}\}\\ \mathit{Mod}(\texttt{constrH}) = \texttt{cMod}\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{this} : \texttt{X},\quad \mathit{Accessible}(\texttt{D},\texttt{X},\texttt{C},\texttt{cMod})\end{array}}{\texttt{D},\Gamma \vdash_{s/n} \textbf{new}\ \texttt{C}(\texttt{e}_1...\texttt{e}_\texttt{n}) : \texttt{C}}$$

$$\frac{\begin{array}{l}\texttt{D} \vdash \texttt{T} \diamond_{VarType}\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{e}_\texttt{i} : \texttt{int} \quad \texttt{i} \in \{1...\texttt{n}\}, \texttt{n} \geq 1, \texttt{k} \geq 0\end{array}}{\texttt{D},\Gamma \vdash_{s/n} \textbf{new}\ \texttt{T}[\texttt{e}_1]...[\texttt{e}_\texttt{n}][]_1...[]_\texttt{k} : \texttt{T}[]_1...[]_{\texttt{n}+\texttt{k}}}$$

$$\frac{\begin{array}{l}\texttt{D},\Gamma \vdash_{s/n} \texttt{v} : \texttt{T}[]\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{e} : \texttt{int}\end{array}}{\texttt{D},\Gamma \vdash_{s/n} \texttt{v[e]} : \texttt{T}} \qquad \frac{\begin{array}{l}\vdash\ \texttt{D} \diamond_a\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{v} : \texttt{T}\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{this} : \texttt{C}\\ \mathit{Accessible}(\texttt{D},\texttt{C},\texttt{X},\texttt{fMod})\\ \mathit{FirstVis}(\texttt{D},\texttt{T},\texttt{f}) = (\texttt{fMod X T}'\ \texttt{f})\end{array}}{\texttt{D},\Gamma \vdash_{s/n} \texttt{v.f} : \texttt{T}'} \qquad \frac{\begin{array}{l}\vdash\ \texttt{D} \diamond_a\\ \texttt{D} \vdash \texttt{X} \sqsubseteq \texttt{X}\ \text{ or }\ \texttt{D} \vdash \texttt{X} \leq \texttt{X}\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{this} : \texttt{C}\\ \mathit{Accessible}(\texttt{D},\texttt{C},\texttt{X},\texttt{fMod})\\ \mathit{FSpec}(\texttt{D},\texttt{X},\texttt{f}) = (\texttt{fMod T f})\\ \textbf{static} \in \texttt{fMod}\end{array}}{\texttt{D},\Gamma \vdash_{s/n} \texttt{X.f} : \texttt{T}}$$

$$\frac{\begin{array}{l}\vdash\ \texttt{D} \diamond_a\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{e}_\texttt{i} : \texttt{T}_\texttt{i} \quad \texttt{i} \in \{0...\texttt{n}\}, \texttt{n} \geq 0\\ \mathit{MostSpec}(\texttt{D},\texttt{m},\texttt{T}_0,\texttt{T}_1 \times ... \times \texttt{T}_\texttt{n}) = \{\texttt{methH}\}\\ \mathit{Mod}(\texttt{methH}) = \texttt{mMod}\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{this} : \texttt{X},\quad \mathit{Accessible}(\texttt{D},\texttt{X},\texttt{C},\texttt{mMod})\end{array}}{\texttt{D},\Gamma \vdash_{s/n} \texttt{e}_0\texttt{.m}(\texttt{e}_1...\texttt{e}_\texttt{n}) : \mathit{RetT}(\texttt{methH})} \qquad \frac{\begin{array}{l}\texttt{D},\Gamma \vdash_{s/n} \texttt{e} : \texttt{E}\\ \texttt{D} \vdash \texttt{E} \sqsubseteq \texttt{Exception}\end{array}}{\texttt{D},\Gamma \vdash_{s/n} \textbf{throw}\ \texttt{e} : \textbf{void}}$$

$$\frac{\begin{array}{l}\texttt{n} \geq 0\\ \texttt{D} \vdash \texttt{E}_\texttt{i} \sqsubseteq \texttt{Exception} \quad \texttt{i} \in \{1...\texttt{n}\}\\ \texttt{D} \vdash \Gamma \diamond\\ \Gamma(\texttt{v}_\texttt{i}) = \mathit{Undef} \quad \texttt{i} \in \{1...\texttt{n}\}\\ \texttt{D},\Gamma;\ \texttt{E}_\texttt{i}\ \texttt{v}_\texttt{i}\ \vdash_{s/n} \texttt{stmts}_\texttt{i} : \textbf{void} \quad \texttt{i} \in \{1...\texttt{n}\}\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{stmts}_0 : \textbf{void}\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{stmts}_{\texttt{n}+1} : \textbf{void}\end{array}}{\begin{array}{l}\texttt{D},\Gamma \vdash_{s/n} \textbf{try}\ \texttt{stmts}_0\ \textbf{catch}\ \texttt{E}_1\ \texttt{v}_1\ \texttt{stmts}_1\ ...\ \textbf{catch}\ \texttt{E}_\texttt{n}\ \texttt{v}_\texttt{n}\ \texttt{stmts}_\texttt{n} : \textbf{void}\\ \texttt{D},\Gamma \vdash_{s/n} \textbf{try}\ \texttt{stmts}_0\ \textbf{catch}\ \texttt{E}_1\ \texttt{v}_1\ \texttt{stmts}_1\ ...\ \textbf{catch}\ \texttt{E}_\texttt{n}\ \texttt{v}_\texttt{n}\ \texttt{stmts}_\texttt{n}\\ \qquad \textbf{finally}\ \texttt{stmts}_{\texttt{n}+1} : \textbf{void}\end{array}}$$

Figure 16: *Normal types* for Java$_s$ terms

$$\frac{\texttt{i}\text{ is integer,}\quad \texttt{c}\text{ is character,}\quad \Gamma(\texttt{x}) \neq \mathit{Undef}}{\begin{array}{l}\mathtt{D},\Gamma \vdash_{s/e} \mathtt{true} : \emptyset,\quad \mathtt{D},\Gamma \vdash_{s/e} \mathtt{false} : \emptyset \\ \mathtt{D},\Gamma \vdash_{s/e} \mathtt{i} : \emptyset,\quad \mathtt{D},\Gamma \vdash_{s/e} \mathtt{c} : \emptyset,\quad \mathtt{D},\Gamma \vdash_{s/e} \mathtt{x} : \emptyset\end{array}} \qquad \frac{}{\mathtt{D},\Gamma \vdash_{s/e} \mathtt{null} : \mathtt{nil}}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma \vdash_{s/e} \mathtt{e} : \mathtt{ET} \\ \mathtt{D},\Gamma \vdash_{s/e} \mathtt{stmts} : \mathtt{ET}',\quad \mathtt{D},\Gamma \vdash_{s/e} \mathtt{stmts}' : \mathtt{ET}'',\quad \mathtt{D},\Gamma \vdash_{s/e} \mathtt{stmt} : \mathtt{ET}'''\end{array}}{\begin{array}{l}\mathtt{D},\Gamma \vdash_{s/e} \mathtt{stmts;stmt} : \mathtt{ET}' \cup \mathtt{ET}''' \\ \mathtt{D},\Gamma \vdash_{s/e} \textbf{if } \mathtt{e}\ \textbf{ then }\ \mathtt{stmts}\ \textbf{ else } \mathtt{stmts}' : \mathtt{ET} \cup \mathtt{ET}' \cup \mathtt{ET}''\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma \vdash_{s/e} \mathtt{v} : \mathtt{ET} \\ \mathtt{D},\Gamma \vdash_{s/e} \mathtt{e} : \mathtt{ET}'\end{array}}{\mathtt{D},\Gamma \vdash_{s/e} \mathtt{v{=}e}\ :\ \mathtt{ET} \cup \mathtt{ET}'}$$

$$\frac{\mathtt{D},\Gamma \vdash_{s/e} \mathtt{e} : \mathtt{ET}}{\mathtt{D},\Gamma \vdash_{s/e}\ \textbf{return}\ \mathtt{e} : \mathtt{ET}} \qquad \frac{}{\mathtt{D},\Gamma \vdash_{s/e}\ \textbf{return} : \emptyset}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma \vdash_{s/e} \mathtt{e_i} : \mathtt{ET_i}\quad \mathtt{i} \in \{\mathtt{1...n}\},\ \mathtt{n} \geq 1, \\ \mathtt{D},\Gamma \vdash_{s/n} \mathtt{e_i} : \mathtt{T_i}\quad \mathtt{i} \in \{\mathtt{1...n}\},\ \mathtt{n} \geq 1, \\ \mathit{MostSpecConstr}(\mathtt{D},\mathtt{C},\mathtt{T_1} \times ... \times \mathtt{T_n}) = \{\mathtt{constrH}\}\end{array}}{\mathtt{D},\Gamma \vdash_{s/e} \textbf{new } \mathtt{C(e_1...e_n)} : (\cup \mathtt{ET_i})_{\mathtt{i=1}}^{\mathtt{n}} \cup \mathit{ExcT}(\mathtt{constrH})}$$

$$\frac{\mathtt{D},\Gamma \vdash_{s/e} \mathtt{e_i} : \mathtt{ET_i},\qquad \mathtt{i} \in \{\mathtt{1...n}\}}{\mathtt{D},\Gamma \vdash_{s/e} \textbf{new } \mathtt{T[e_1]...[e_n][]_1...[]_k} : (\cup \mathtt{ET_i})_{\mathtt{i=1}}^{\mathtt{n}}}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma \vdash_{s/e} \mathtt{v} : \mathtt{ET} \\ \mathtt{D},\Gamma \vdash_{s/e} \mathtt{e} : \mathtt{ET}'\end{array}}{\mathtt{D},\Gamma \vdash_{s/e} \mathtt{v[e]} : \mathtt{ET} \cup \mathtt{ET}'} \qquad \frac{\mathtt{D},\Gamma \vdash_{s/e} \mathtt{v} : \mathtt{ET}}{\mathtt{D},\Gamma \vdash_{s/e} \mathtt{v.f} : \mathtt{ET}} \qquad \frac{\begin{array}{l}\mathtt{D} \vdash \mathtt{X} \sqsubseteq \mathtt{X} \\ \quad \text{or}\ \ \mathtt{D} \vdash \mathtt{X} \leq \mathtt{X}\end{array}}{\mathtt{D},\Gamma \vdash_{s/e} \mathtt{X.f} : \emptyset}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma \vdash_{s/e} \mathtt{e_i} : \mathtt{ET_i} \\ \mathtt{D},\Gamma \vdash_{s/n} \mathtt{e_i} : \mathtt{T_i}\quad \mathtt{i} \in \{\mathtt{0...n}\},\ \mathtt{n} \geq 0, \\ \mathit{MostSpec}(\mathtt{D},\mathtt{m},\mathtt{T_0},\mathtt{T_1} \times ... \times \mathtt{T_n}) = \{\mathtt{methH}\}\end{array}}{\mathtt{D},\Gamma \vdash_{s/e} \mathtt{e_0.m(e_1...e_n)} : (\cup \mathtt{ET_i})_{\mathtt{i=0}}^{\mathtt{n}} \cup \mathit{ExcT}(\mathtt{methH})} \qquad \frac{\begin{array}{l}\mathtt{D},\Gamma \vdash_{s/n} \mathtt{e} : \mathtt{E}, \\ \mathtt{D},\Gamma \vdash_{s/e} \mathtt{e} : \mathtt{ET}\end{array}}{\mathtt{D},\Gamma \vdash_{s/e} \textbf{throw } \mathtt{e} : \{\mathtt{E}\} \cup \mathtt{ET}}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0 \\ \mathtt{D} \vdash \mathtt{E_i}\ \sqsubseteq \mathtt{Exception}\quad \mathtt{i} \in \{\mathtt{1...n}\} \\ \mathtt{D} \vdash \Gamma \diamond \\ \Gamma(\mathtt{v_i}) = \mathit{Undef}\quad \mathtt{i} \in \{\mathtt{1...n}\} \\ \mathtt{D},\Gamma;\ \mathtt{E_i}\ \mathtt{v_i}\ \vdash_{s/e} \mathtt{stmts_i} : \mathtt{ET_i}\quad \mathtt{i} \in \{\mathtt{1...n}\} \\ \mathtt{D},\Gamma \vdash_{s/e} \mathtt{stmts_0} : \mathtt{ET_0} \\ \mathtt{D},\Gamma \vdash_{s/e} \mathtt{stmts_{n+1}} : \mathtt{ET_{n+1}}\end{array}}{\begin{array}{l}\mathtt{D},\Gamma \vdash_{s/e} \textbf{try } \mathtt{stmts_0}\ \textbf{catch } \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}...\ \textbf{catch } \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ : \\ \qquad\qquad\qquad\qquad (\mathtt{ET_0} \setminus_{\mathtt{D}}^{\mathtt{e}} \{\mathtt{E_1},\ldots,\mathtt{E_n}\}) \cup (\cup \mathtt{ET_i})_{\mathtt{i=1}}^{\mathtt{n}} \\ \mathtt{D},\Gamma \vdash_{s/e} \textbf{try } \mathtt{stmts_0}\ \textbf{catch } \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}...\ \textbf{catch } \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n} \\ \qquad \textbf{finally}\ \ \mathtt{stmts_{n+1}}\ :\ (\mathtt{ET_0} \setminus_{\mathtt{D}}^{\mathtt{e}} \{\mathtt{E_1},\ldots,\mathtt{E_n}\}) \cup (\cup \mathtt{ET_i})_{\mathtt{i=1}}^{\mathtt{n+1}}\end{array}}$$

Figure 17: *Abnormal types* for Java$_s$ terms

$$
\begin{array}{lll}
\mathtt{P}, \Gamma \vdash_{s} \mathtt{t} : \tau & \textit{iff} & \mathtt{D}, \Gamma \vdash_{s} \mathtt{t} : \tau; \\
\mathtt{P}, \Gamma \vdash_{s/n} \mathtt{t} : \mathtt{T} & \textit{iff} & \mathtt{D}, \Gamma \vdash_{s/n} \mathtt{t} : \mathtt{T}; \\
\mathtt{P}, \Gamma \vdash_{s/e} \mathtt{t} : \mathtt{ET} & \textit{iff} & \mathtt{D}, \Gamma \vdash_{s/e} \mathtt{t} : \mathtt{ET}
\end{array}
$$

3.5 Java$_{se}$ – extended

Java$_{se}$ is an enriched version of Java$_s$ which provides compile-time type information necessary at run-time. The syntax of Java$_{se}$ programs is defined in figure 19. The process of enriching Java$_s$ terms is described by the mapping $\mathcal{C}$.

The fundamental program $\mathtt{P}_{\mathtt{FUIO}/se}$ corresponding to that $\mathtt{P}_{\mathtt{FUIO}/s}$, given for Java$_s$ in section 1, is defined in figure 18. It contains the same classes and interfaces as $\mathtt{P}_{\mathtt{FUIO}/s}$ which are enriched and therefore different from those in $\mathtt{P}_{\mathtt{FUIO}/s}$.

$\mathtt{P}_{\mathtt{FUIO}/se}$ = **class** `Object` f ...g
class `Exception` **ext** `Object` f ...g
class `RTE` **ext** `Exception` f ...g
class `NullPE` **ext** `RTE` f ...g
class `ArrStoreE` **ext** `RTE` f ...g
class `IndOutBndE` **ext** `RTE` f ...g
class `NegSzeE` **ext** `RTE` f ...g ...

Figure 18: The Java$_{se}$ fundamental program

There are three cases where Java$_{se}$ is different from Java$_s$. Method calls are enriched by the *argument types* of the most special applicable method available at compile-time. Thus, the Java$_s$ syntax *Expr*.`MethName(`*Expr**`)` is replaced in Java$_{se}$ by *Expr*.[*ArgType*] `MethName(`*Expr**`)`. Instance variable accesses are enriched by a descriptor – the class containing the field declaration. *Expr*.`VarName` is replaced in Java$_{se}$ by *Expr*.`[ClassName]VarName`. Finally the `new C()` expression is enriched by the argument type of the most special constructor available at compile time.

Examples of enriching method call and instance variable access can be found in the following $\mathtt{P}_{\mathtt{Phil}/se}$ program, which would be the result of the mapping $\mathcal{C}$ the program $\mathtt{P}_{\mathtt{Animal}/s}$.

$\mathtt{P}_{\mathtt{Animal}/se} = \mathcal{C}\{\mathtt{D}_{\mathtt{FUIO}}\ \mathtt{D}_{\mathtt{Animal}}, \epsilon, \mathtt{P}_{\mathtt{Animal}/s}\} =$
abstract class `Animal` **ext** `Object` **impl** $\emptyset$ f
  protected boolean `isHealthy ;`
  `Animal()` **throws** f /∗ `animalConstrBody`$_1$ ∗/ g
  `Animal(` **boolean** `healthy )` **throws** f ...[]**this**`();` **this**.[`Animal`]`isHealthy` ... g
  void `move()` **throws** f /∗ `animalMoveBody`$_1$ ∗/ g
  protected abstract boolean `health()` **throws** ;
g

final class `Cat` **ext** `Animal` **impl** `Carnivore` f
  public int `numLives ;`
  static int `numCatsAlive ;`
  public void `attack(Cat c)` **throws** f /∗ `attackBody`$_1$ ∗/ g
  private void `attack(Dog d)` **throws** f /∗ `attackBody`$_2$ ∗/ g
  public void `eat(Cat c)` **throws** f `Cat.numCatsAlive` ... g
  void `move()` **throws** f /∗ `catMoveBody`$_1$ ∗/ g
  public boolean `health()` **throws** f /∗ `catHealthBody`$_1$ ∗/ g
g

class `Dog` **extends** `Animal` **impl** $\emptyset$ f
  public final int `numLegs ;`

Program	::=	Def^*
Def	::=	*ClassMod* **class** `ClassId` **ext** `ClassName` **impl** `InterfName`*
		f $Constructor^*$ $ClassMember^*$g
	|	*InterfMod* **interface** `InterfId` **ext** `InterfName`*
		f $InterfMember^*$g
ClassMod	::=	[**public**] [**abstract**| **final**]
InterfMod	::=	[**public**] **abstract**
AccessMod	::=	**public** | **protected** | **private** | ϵ
ConstrMod	::=	*AccessMod*
FieldMod	::=	*AccessMod* [**static**] [**final**]
IFieldMod	::=	**public static final**
MethMod	::=	*AccessMod* [**final**| **abstract**]
Constructor	::=	*ConstrHeader* *ConstrBody*
ConstrHeader	::=	*ConstrMod* `ConstrId`((*VarType*`ParId`)*)
		throws `ClassName`*
ConstrBody	::=	f *ConstrInvoke*; *Stmts*; **return**g
ConstrInvoke	::=	`this` ($Expr^*$) | `super` ($Expr^*$)
ClassMember	::=	*Field* | *Method*
InterfMember	::=	*IField*
Field	::=	*FieldMod* *VarType* `VarId` [*Expr*] ;
IField	::=	*IFieldMod* *VarType* `VarId` *Expr* ;
Method	::=	*MethHeader* *MethBody*
MethHeader	::=	*MethMod* (**void** | *VarType*) `MethId` ((*VarType* `ParId`)*)
		throws `ClassName`*
MethBody	::=	f *Stmts* ; **return** [*Expr*] g
Stmts	::=	ϵ | *Stmts* ; *Stmt*
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** [*ArgType*]`ClassName` ($Expr^*$)
	|	**new** *SimpleType* ([*Expr*])$^+$([])*
Var	::=	`Name` | *RecvExpr*.[`ClassName`]`VarName` | *Var* [*Expr*] | **this**
RecvExpr	::=	*Expr* | `ClassName` | `InterfaceName`
Value	::=	*PrimValue* | `null`
PrimValue	::=	*intValue* | *charValue* | *boolValue* | ...
VarType	::=	*SimpleType* | *ArrayType*
SimpleType	::=	*PrimType* | `ClassName` | `InterfaceName`
ArrayType	::=	*SimpleType* [] | *ArrayType* []
PrimType	::=	**bool** | **char** | **int** | ...

Figure 19: Java$_{se}$ programs

```
        public int siblings ;
        public boolean isHealthy ;
        Dog()   throws   f  [int]this(0);  g
        private Dog( int  sib )   throws   f  []this();  this.[Dog]siblings ... ;  g
        Dog( boolean  healthy )   throws   f  [boolean]super(healthy);  this.[Dog]isHealthy ...  g
        public  boolean  health()   throws   f /* dogHealthBody1 */ g
g

interface Carnivore   ext  ∅ f
        public static final Cat  diet;
        public abstract void eat(Cat  c)   throws    ;
g
```

A particular difference between $\mathtt{P}_{\mathtt{Animal}/s}$ and $\mathtt{P}_{\mathtt{Animal}/se}$ is that access to the instance variable `isHealthy` in the constructor `Dog(`**boolean** `healthy)(...)` from the class `Dog` is enriched with the name of the class `Dog` containing field definition (**public boolean** `isHealthy = true;`), which is first visible in the class hierarchy.
The function $\mathcal{D}$ is redefined for Java$_{se}$ programs in the expected way. The resulting declarations coincide with those originally defined.

3.5.1 Enriching / Compilation – extended

The compilation of Java_s programs is described by the mapping $\mathcal{C}$ from a triplet, consisting of declaration with all the information required, environment, where the compilation occurs, and a Java_s program itself, to the enriched, Java_{se}, program:

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

The compilation of Java_s programs is described separately from type checking by the inference rules in figure 20.

According to the first rule, compilation of literals leaves them unmodified. Compilation of all other constructs consists of the compilation of constituent components with the exception of field access (sixth line) and method call (seventh line). These are described in detail below. According to the rule on the second line, the compilation of statement sequence consists of the consequent compilation of the statements. According to the rule on the third line, the compilation of assignment statement consists of compilation of the variable and expression. According to the rules on the fourth line, the compilation of return statement depends only on the expression it returns if there is an expression.

The fifth line describes the compilation of a constructor invocation expression. The expression is enriched with the *argument type* of the most special applicable constructor *constr* available in compile-time as found by the type checker through definition 13.

The first rule on the sixth line describes the compilation of array creation. For a simple type which is primitive type or a class (but not interface) the compilation of the expression **new** $\mathtt{T[e_1]...[e_n][]_1...[]_k}$ depends on the compilation of the expression for calculating the array demension bounds and enriches the compiled expression with initial values of type `T` for the component initialization.

The second rule of the sixth line and the seventh line describes the compilation of field and array access. Compilation of the array access `v[e]` consists of the compilation of the variable `v` and expression `e`. Compilation of field access `v.f` extends the expression by the descriptor `C`, the name of the class from which the field definition is inherited, unless the `f` is determined to have been declared **static** in `C` in which case the expression `v` is simply replaced by `C`. This information about `C` is obtained using the definition 14.

In line eight, the compilation results in the enrichment of the method call expression by the *argument type* of the most special applicable method *meth* available in compile-time in the class `T` as defined in definition 15.

Note that for field access and method call we require the program to be acyclic in order to guarantee that the look up functions are well-formed.

In the remaining two rules, the compilation of **throw** statement depends on the expression it throws. The compilation of **try** ... **catch** ... **finally** statements consists of the compilation of the corresponding statements from the **try** block, all **catch** blocks and from the **finally** block if it exists.

$$\frac{}{\mathcal{C}\{\texttt{D},\Gamma,\texttt{z}\} = \texttt{z} \qquad \text{if } \texttt{z} \text{ is integer, character, identifier, } \texttt{null}, \texttt{true}, \text{ or } \texttt{false}}$$

$$\frac{}{\begin{array}{l}\mathcal{C}\{\texttt{D},\Gamma,\texttt{stmts ; stmt}\} = \mathcal{C}\{\texttt{D},\Gamma,\texttt{stmts}\}\ \texttt{;}\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{stmt}\}\\ \mathcal{C}\{\texttt{D},\Gamma,\textbf{if}\ \texttt{e}\ \textbf{then}\ \texttt{stmts}\ \textbf{else}\ \texttt{stmts}'\} = \\ \quad \textbf{if}\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{e}\}\ \textbf{then}\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{stmts}\}\ \textbf{else}\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{stmts}'\}\end{array}}$$

$$\frac{}{\mathcal{C}\{\texttt{D},\Gamma,\texttt{v=e}\} = \mathcal{C}\{\texttt{D},\Gamma,\texttt{v}\}\texttt{=}\mathcal{C}\{\texttt{D},\Gamma,\texttt{e}\}}$$

$$\frac{}{\mathcal{C}\{\texttt{D},\Gamma,\ \textbf{return}\ \texttt{e}\} = \textbf{return}\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{e}\}} \qquad \frac{}{\mathcal{C}\{\texttt{D},\Gamma,\ \textbf{return}\} = \textbf{return}}$$

$$\frac{\begin{array}{l}\texttt{D} \vdash \texttt{C} \sqsubseteq \texttt{C}\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{e}_\texttt{i} : \texttt{T}_\texttt{i} \quad \texttt{i} \in \{\texttt{1}...\texttt{n}\}, \texttt{n} \geq 1\\ \mathit{MostSpecConstr}(\texttt{D},\texttt{C},\texttt{T}_1 \times ... \times \texttt{T}_\texttt{n}) = \{\texttt{constrH}\}\end{array}}{\begin{array}{l}\mathcal{C}\{\texttt{D},\Gamma,\textbf{new}\ \texttt{C}(\texttt{e}_1...\texttt{e}_\texttt{n})\} = \\ \quad \textbf{new}\ [\mathit{ArgT}(\texttt{constrH})]\texttt{C}(\mathcal{C}\{\texttt{D},\Gamma,\texttt{e}_1\}\ ...\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{e}_\texttt{n}\})\end{array}}$$

$$\frac{\texttt{D} \vdash \texttt{T} \diamond_{VarType}}{\begin{array}{l}\mathcal{C}\{\texttt{D},\Gamma,\textbf{new}\ \texttt{T}[\texttt{e}_1]...[\texttt{e}_\texttt{n}][]_1...[]_\texttt{k}\} = \\ \quad \textbf{new}\ \texttt{T}[\mathcal{C}\{\texttt{D},\Gamma,\texttt{e}_1\}\Gamma]...[\mathcal{C}\{\texttt{D},\Gamma,\texttt{e}_\texttt{n}\}\Gamma][]_1...[]_\texttt{k}\end{array}} \qquad \frac{\begin{array}{l}\vdash \texttt{D} \diamond_a\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{v} : \texttt{T}\\ \mathit{FirstVis}(\texttt{D},\texttt{T},\texttt{f}) = (\texttt{fMod X T}'\ \texttt{f})\\ \textbf{static} \in \texttt{fMod}\end{array}}{\mathcal{C}\{\texttt{D},\Gamma,\texttt{v.f}\} = \texttt{X.f}}$$

$$\frac{}{\mathcal{C}\{\texttt{D},\Gamma,\texttt{v[e]}\} = \mathcal{C}\{\texttt{D},\Gamma,\texttt{v}\}[\mathcal{C}\{\texttt{D},\Gamma,\texttt{e}\}]} \qquad \frac{\begin{array}{l}\vdash \texttt{D} \diamond_a\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{v} : \texttt{T}\\ \mathit{FirstVis}(\texttt{D},\texttt{T},\texttt{f}) = (\texttt{fMod X T}'\ \texttt{f})\\ \textbf{static} \notin \texttt{fMod}\end{array}}{\mathcal{C}\{\texttt{D},\Gamma,\texttt{v.f}\} = \mathcal{C}\{\texttt{D},\Gamma,\texttt{v}\}.[\texttt{X}]\texttt{f}}$$

$$\frac{\begin{array}{l}\vdash \texttt{D} \diamond_a\\ \texttt{D},\Gamma \vdash_{s/n} \texttt{e}_\texttt{i} : \texttt{T}_\texttt{i} \quad \texttt{i} \in \{\texttt{0}...\texttt{n}\}, \texttt{n} \geq 0\\ \mathit{MostSpec}(\texttt{D},\texttt{m},\texttt{T}_0,\texttt{T}_1 \times ... \times \texttt{T}_\texttt{n}) = \{\texttt{methH}\}\end{array}}{\begin{array}{l}\mathcal{C}\{\texttt{D},\Gamma,\texttt{e}_0.\texttt{m}(\texttt{e}_1...\texttt{e}_\texttt{n})\} = \\ \quad \mathcal{C}\{\texttt{D},\Gamma,\texttt{e}_0\}.[\mathit{ArgT}(\texttt{methH})]\texttt{m}(\mathcal{C}\{\texttt{D},\Gamma,\texttt{e}_1\}...\mathcal{C}\{\texttt{D},\Gamma,\texttt{e}_\texttt{n}\})\end{array}}$$

$$\frac{}{\mathcal{C}\{\texttt{D},\Gamma,\textbf{throw}\ \texttt{e}\} = \textbf{throw}\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{e}\}}$$

$$\frac{}{\begin{array}{l}\mathcal{C}\{\texttt{D},\Gamma,\textbf{try}\ \texttt{stmts}_0\ \textbf{catch}\ \texttt{E}_1\ \texttt{v}_1\ \texttt{stmts}_1\ ...\ \textbf{catch}\ \texttt{E}_\texttt{n}\ \texttt{v}_\texttt{n}\ \texttt{stmts}_\texttt{n}\}\\ \quad = \textbf{try}\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{stmts}_0\}\ \textbf{catch}\ \texttt{E}_1\ \texttt{v}_1\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{stmts}_1\}\ ...\\ \qquad \textbf{catch}\ \texttt{E}_\texttt{n}\ \texttt{v}_\texttt{n}\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{stmts}_\texttt{n}\}\\ \mathcal{C}\{\texttt{D},\Gamma,\textbf{try}\ \texttt{stmts}_0\ \textbf{catch}\texttt{E}_1\texttt{v}_1\texttt{stmts}_1...\textbf{catch}\texttt{E}_\texttt{n}\ \texttt{v}_\texttt{n}\ \texttt{stmts}_\texttt{n}\ \textbf{finally}\ \texttt{stmts}_{\texttt{n}+1}\}\\ \quad = \textbf{try}\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{stmts}_0\}\ \textbf{catch}\ \texttt{E}_1\ \texttt{v}_1\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{stmts}_1\}\ ...\\ \qquad \textbf{catch}\ \texttt{E}_\texttt{n}\ \texttt{v}_\texttt{n}\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{stmts}_\texttt{n}\}\ \textbf{finally}\ \mathcal{C}\{\texttt{D},\Gamma,\texttt{stmts}_{\texttt{n}+1}\}\end{array}}$$

Figure 20: Compilation of Java_s terms

3.6 Java$_{se}$ types

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

The type rules for the compiled and enriched Java$_{se}$ term $\mathtt{t}$ have the following forms:

- $\mathtt{D}, \Gamma \vdash_{se/n} \mathtt{t} : \mathtt{T}$ – for the *normal type*;
- $\mathtt{D}, \Gamma \vdash_{se/e} \mathtt{t} : \mathtt{ET}$ – for the *abnormal type*;
- $\mathtt{D}, \Gamma \vdash_{se} \mathtt{t} : \mathtt{T} \,||\, \mathtt{ET}$ – for the *type*.

Note the subscripts under the $\vdash$ which is used in order to distinguish various type systems and indicate the *normal type* and *abnormal type* inference in the Java$_{se}$ language.
The type rules for Java$_{se}$ terms are given for the *normal type* in figure 21 and for the *abnormal type* in figure 22.

3.6.1 Normal types of Java$_{se}$

Description of the normal types of Java$_{se}$terms

Figure 21 describes the *normal types* for Java$_{se}$ terms. Figure 21 and figure 16 have identical structure. There are only four rules which are different. These are field access and method call, and the object creation expression and the **static** field access using the qualified name.

The second rule on the sixth line describes field access. The difference between the *normal type* of a field access expression in Java$_s$ and Java$_{se}$ is that in Java$_{se}$ the *normal type* depends on the descriptor (*i.e.* $\mathtt{C}$) instead of the *normal type* of the variable on the left of the field access (*i.e.* $\mathtt{T}$).

On the seventh line we consider Java$_{se}$ method calls: we search for appropriate methods using the *argument type* $(\mathtt{T_1} \times ... \times \mathtt{T_n})$ instead of the types of the actual expressions $(\mathtt{T'_1}, ...\mathtt{T'_n})$. 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 18 *For a program specification* $\mathtt{D}$ *satisfying* $\vdash \mathtt{D} \diamond_a$*, identifier* $\mathtt{m}$*, variable type* $\mathtt{T}$*, argument type* $\mathtt{AT}$*, we define:*

$$FirstFit(\mathtt{D}, \mathtt{m}, \mathtt{T}, \mathtt{AT}) = \{\mathtt{methH} \mid \mathtt{methH} \in Meths(\mathtt{D}, \mathtt{T}, \mathtt{m}) \;\; and \;\; ArgT(\mathtt{methH}) = \mathtt{AT}\}$$

Definition 19 *For a program specification* $\mathtt{D}$ *satisfying* $\vdash \mathtt{D} \diamond_a$*, class* $\mathtt{C}$*, argument type* $\mathtt{AT}$*, we define:*

$$FirstFitConstr(\mathtt{D}, \mathtt{C}, \mathtt{AT}) = \{\mathtt{constrH} \mid \mathtt{constrH} \in ConstrSpec(\mathtt{D}, \mathtt{C}) \;\; and \;\; ArgT(\mathtt{constrH}) = \mathtt{AT}\}$$

3.6.2 Abnormal types of Java$_{se}$

Description of the abnormal types of Java$_{se}$ terms Figure 22 describes the *abnormal types* for Java$_{se}$ terms. The rules for *abnormal types* for Java$_{se}$ terms are identical to those for Java$_s$ given in figure 17. The differences occur only in the syntax of field access, and **static** field access, new object creation expression and method call.

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

Definition 20 $\mathtt{D}, \Gamma \vdash_{se} \mathtt{t} : \mathtt{T} \,||\, \mathtt{ET}$ *iff* $\mathtt{D}, \Gamma \vdash_{se/n} \mathtt{t} : \mathtt{T}$ *and* $\mathtt{D}, \Gamma \vdash_{se/e} \mathtt{t} : \mathtt{ET}$

$$\frac{\mathtt{i}\text{ is integer},\quad \mathtt{c}\text{ is character},\quad \Gamma(\mathtt{x}) = \mathtt{T\ x}}{\begin{array}{l}\mathtt{D},\Gamma \vdash_{se/n} \mathtt{true} : \mathbf{bool},\quad \mathtt{D},\Gamma \vdash_{se/n} \mathtt{false} : \mathbf{bool},\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{i} : \mathbf{int},\quad \mathtt{D},\Gamma \vdash_{se/n} \mathtt{c} : \mathbf{char},\quad \mathtt{D},\Gamma \vdash_{se/n} \mathtt{x} : \mathtt{T}\end{array}} \qquad \frac{}{\mathtt{D},\Gamma \vdash_{se/n} \mathtt{null} : \mathtt{nil}}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma \vdash_{se/n} \mathtt{e} : \mathbf{bool}\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{stmts} : \mathbf{void} \qquad \mathtt{D},\Gamma \vdash_{se/n} \mathtt{stmts}' : \mathbf{void} \qquad \mathtt{D},\Gamma \vdash_{se/n} \mathtt{stmt} : \mathtt{T}\end{array}}{\begin{array}{l}\mathtt{D},\Gamma \vdash_{se/n} \mathtt{stmts}\ ;\ \mathtt{stmt} : \mathtt{T}\\ \mathtt{D},\Gamma \vdash_{se/n} \mathbf{if}\ \mathtt{e}\ \ \mathbf{then}\ \ \mathtt{stmts}\ \ \mathbf{else}\ \mathtt{stmts}' : \mathbf{void}\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma \vdash_{se/n} \mathtt{v} : \mathtt{T}\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{e} : \mathtt{T}'\\ \mathtt{D} \vdash \mathtt{T}' \leq_w \mathtt{T}\\ \mathit{NonFinalField}(\mathtt{D},\mathtt{v})\end{array}}{\mathtt{D},\Gamma \vdash_{se/n} \mathtt{v{=}e} : \mathbf{void}} \qquad \frac{\mathtt{D},\Gamma \vdash_{se/n} \mathtt{e} : \mathtt{T}}{\mathtt{D},\Gamma \vdash_{se/n} \mathbf{return}\ \mathtt{e} : \mathtt{T}} \qquad \frac{}{\mathtt{D},\Gamma \vdash_{se/n} \mathbf{return} : \mathbf{void}}$$

$$\frac{\begin{array}{l}\mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C},\quad \mathit{NonAbstractClass}(\mathtt{D},\mathtt{C})\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{e_i} : \mathtt{T_i}\quad \mathtt{i} \in \{\mathtt{1...n}\}, \mathtt{n} \geq 1\\ \mathit{FirstFitConstr}(\mathtt{D},\mathtt{C},\mathtt{T_1} \times ... \times \mathtt{T_n}) = \{\mathtt{constrH}\}\\ \mathit{Mod}(\mathtt{constrH}) = \mathtt{cMod}\\ \mathtt{D},\Gamma \vdash_{s/n} \mathtt{this} : \mathtt{X},\quad \mathit{Accessible}(\mathtt{D},\mathtt{X},\mathtt{C},\mathtt{cMod})\end{array}}{\mathtt{D},\Gamma \vdash_{se/n} \mathbf{new}\ [\mathtt{T_1} \times ... \times \mathtt{T_n}]\mathtt{C}(\mathtt{e_1}...\mathtt{e_n}) : \mathtt{C}}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 1, \mathtt{k} \geq 0\\ \mathtt{D} \vdash \mathtt{T} \diamond_{VarType}\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{e_i} : \mathtt{int}\quad \mathtt{i} \in \{\mathtt{1...n}\}\end{array}}{\mathtt{D},\Gamma \vdash_{se/n} \mathbf{new}\ \mathtt{T[e_1]...[e_n][]_1...[]_k} : \mathtt{T[]_1...[]_{n+k}}}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma \vdash_{se/n} \mathtt{v} : \mathtt{T[]}\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{e} : \mathtt{int}\end{array}}{\mathtt{D},\Gamma \vdash_{se/n} \mathtt{v[e]} : \mathtt{T}} \qquad \frac{\begin{array}{l}\vdash\ \mathtt{D} \diamond_a\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{v} : \mathtt{T}\\ \mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{X}\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{this} : \mathtt{C}\\ \mathit{Accessible}(\mathtt{D},\mathtt{C},\mathtt{X},\mathtt{fMod})\\ \mathit{FirstVis}(\mathtt{D},\mathtt{T},\mathtt{f}) = (\mathtt{fMod\ X\ T'\ f})\end{array}}{\mathtt{D},\Gamma \vdash_{se/n} \mathtt{v.[X]f} : \mathtt{T}'} \qquad \frac{\begin{array}{l}\vdash\ \mathtt{D} \diamond_a\\ \mathtt{D} \vdash \mathtt{X} \sqsubseteq \mathtt{X}\ \text{ or }\ \mathtt{D} \vdash \mathtt{X} \leq \mathtt{X}\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{this} : \mathtt{C}\\ \mathit{Accessible}(\mathtt{D},\mathtt{C},\mathtt{X},\mathtt{fMod})\\ \mathit{FSpec}(\mathtt{D},\mathtt{X},\mathtt{f}) = (\mathtt{fMod\ T\ f})\\ \mathbf{static} \in \mathtt{fMod}\end{array}}{\mathtt{D},\Gamma \vdash_{se/n} \mathtt{X.f} : \mathtt{T}}$$

$$\frac{\begin{array}{l}\vdash\ \mathtt{D} \diamond_a\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{e_i} : \mathtt{T}'_\mathtt{i}\quad \mathtt{i} \in \{\mathtt{0...n}\}, \mathtt{n} \geq 0\\ \mathtt{D} \vdash \mathtt{T}'_\mathtt{i} \leq_w \mathtt{T_i}\quad \mathtt{i} \in \{\mathtt{1...n}\}\\ \mathit{FirstFit}(\mathtt{D},\mathtt{m},\mathtt{T}'_0,\mathtt{T_2} \times ... \times \mathtt{T_n}) = \{\mathtt{methH}\}\end{array}}{\mathtt{D},\Gamma \vdash_{se/n} \mathtt{e_0}.[\mathtt{T_1} \times ... \times \mathtt{T_n}\ ]\mathtt{m}(\mathtt{e_1}...\mathtt{e_n}) : \mathit{RetT}(\mathtt{methH})} \qquad \frac{\begin{array}{l}\mathtt{D},\Gamma \vdash_{se/n} \mathtt{e} : \mathtt{E}\\ \mathtt{D} \vdash \mathtt{E} \sqsubseteq \mathtt{Exception}\end{array}}{\mathtt{D},\Gamma \vdash_{se/n} \mathbf{throw}\ \mathtt{e} : \mathbf{void}}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0\\ \mathtt{D} \vdash \mathtt{E_i} \sqsubseteq \mathtt{Exception}\quad \mathtt{i} \in \{\mathtt{1...n}\}\\ \mathtt{D} \vdash \Gamma \diamond\\ \Gamma(\mathtt{v_i}) = \mathit{Undef}\quad \mathtt{i} \in \{\mathtt{1...n}\}\\ \mathtt{D},\Gamma;\ \mathtt{E_i}\ \mathtt{v_i}\ \vdash_{se/n} \mathtt{stmts_i} : \mathbf{void}\quad \mathtt{i} \in \{\mathtt{1...n}\}\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{stmts_0} : \mathbf{void}\\ \mathtt{D},\Gamma \vdash_{se/n} \mathtt{stmts_{n+1}} : \mathbf{void}\end{array}}{\begin{array}{l}\mathtt{D},\Gamma \vdash_{se/n} \mathbf{try}\ \mathtt{stmts_0}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n} : \mathbf{void}\\ \mathtt{D},\Gamma \vdash_{se/n} \mathbf{try}\ \mathtt{stmts_0}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\\ \qquad\qquad \mathbf{finally}\ \ \mathtt{stmts_{n+1}}\ :\ \mathbf{void}\end{array}}$$

Figure 21: *Normal types* for Java$_{se}$ terms

$$\frac{\texttt{i is integer}, \quad \texttt{c is character}, \quad \Gamma(\texttt{x}) \neq \mathit{Undef}}{\begin{array}{l}\texttt{D},\Gamma \vdash_{se/e} \textbf{true} : \emptyset, \quad \texttt{D},\Gamma \vdash_{se/e} \textbf{false} : \emptyset \\ \texttt{D},\Gamma \vdash_{se/e} \texttt{i} : \emptyset, \quad \texttt{D},\Gamma \vdash_{se/e} \texttt{c} : \emptyset, \quad \texttt{D},\Gamma \vdash_{se/e} \texttt{x} : \emptyset\end{array}} \qquad \frac{}{\texttt{D},\Gamma \vdash_{se/e} \texttt{null} : \texttt{nil}}$$

$$\frac{\begin{array}{l}\texttt{D},\Gamma \vdash_{se/e} \texttt{e} : \texttt{ET} \\ \texttt{D},\Gamma \vdash_{se/e} \texttt{stmts} : \texttt{ET}', \quad \texttt{D},\Gamma \vdash_{se/e} \texttt{stmts}' : \texttt{ET}'', \quad \texttt{D},\Gamma \vdash_{se/e} \texttt{stmt} : \texttt{ET}'''\end{array}}{\begin{array}{l}\texttt{D},\Gamma \vdash_{se/e} \texttt{stmts};\texttt{stmt} : \texttt{ET}'\cup\texttt{ET}''' \\ \texttt{D},\Gamma \vdash_{se/e} \textbf{if } \texttt{e} \textbf{ then } \texttt{stmts} \textbf{ else } \texttt{stmts}' : \texttt{ET}\cup\texttt{ET}'\cup\texttt{ET}''\end{array}}$$

$$\frac{\begin{array}{l}\texttt{D},\Gamma \vdash_{se/e} \texttt{v} : \texttt{ET} \\ \texttt{D},\Gamma \vdash_{se/e} \texttt{e} : \texttt{ET}'\end{array}}{\texttt{D},\Gamma \vdash_{se/e} \texttt{v=e} : \texttt{ET}\cup\texttt{ET}'}$$

$$\frac{\texttt{D},\Gamma \vdash_{se/e} \texttt{e} : \texttt{ET}}{\texttt{D},\Gamma \vdash_{se/e} \textbf{return } \texttt{e} : \texttt{ET}} \qquad \frac{}{\texttt{D},\Gamma \vdash_{se/e} \textbf{return} : \emptyset}$$

$$\frac{\begin{array}{l}\texttt{D},\Gamma \vdash_{se/e} \texttt{e}_\texttt{i} : \texttt{ET}_\texttt{i} \quad \texttt{i}\in\{\texttt{1...n}\}, \ \texttt{n} \geq 1, \\ \texttt{D},\Gamma \vdash_{se/n} \texttt{e}_\texttt{i} : \texttt{T}_\texttt{i} \quad \texttt{i}\in\{\texttt{1...n}\}, \ \texttt{n} \geq 1, \\ \mathit{FirstFitConstr}(\texttt{D},\texttt{C},\texttt{T}_1 \times ... \times \texttt{T}_\texttt{n}) = \{\texttt{constrH}\}\end{array}}{\texttt{D},\Gamma \vdash_{se/e} \textbf{new } [\texttt{T}_1 \times ... \times \texttt{T}_\texttt{n}]\texttt{C}(\texttt{e}_1...\texttt{e}_\texttt{n}) : (\cup\texttt{ET}_\texttt{i})_{\texttt{i=1}}^{\texttt{n}} \cup \mathit{ExcT}(\texttt{constrH})}$$

$$\frac{\texttt{D},\Gamma \vdash_{se/e} \texttt{e}_\texttt{i} : \texttt{ET}_\texttt{i} \quad \texttt{i}\in\{\texttt{1...n}\}}{\texttt{D},\Gamma \vdash_{se/e} \textbf{new } \texttt{T}[\texttt{e}_1]...[\texttt{e}_\texttt{n}][]_1...[]_\texttt{k} : (\cup\texttt{ET}_\texttt{i})_{\texttt{i=1}}^{\texttt{n}}}$$

$$\frac{\begin{array}{l}\texttt{D},\Gamma \vdash_{se/e} \texttt{v} : \texttt{ET} \\ \texttt{D},\Gamma \vdash_{se/e} \texttt{e} : \texttt{ET}'\end{array}}{\texttt{D},\Gamma \vdash_{se/e} \texttt{v[e]} : \texttt{ET} \cup \texttt{ET}'} \qquad \frac{\texttt{D},\Gamma \vdash_{se/e} \texttt{v} : \texttt{ET}}{\texttt{D},\Gamma \vdash_{se/e} \texttt{v.[C]f} : \texttt{ET}} \qquad \frac{\begin{array}{c}\texttt{D} \vdash \texttt{X} \sqsubseteq \texttt{X} \\ \text{or } \ \texttt{D} \vdash \texttt{X} \leq \texttt{X}\end{array}}{\texttt{D},\Gamma \vdash_{se/e} \texttt{X.f} : \emptyset}$$

$$\frac{\begin{array}{l}\vdash \ \texttt{D} \diamond_a \\ \texttt{D},\Gamma \vdash_{se/n} \texttt{e}_0 : \texttt{T}_0' \\ \mathit{FirstFit}(\texttt{D},\texttt{m},\texttt{T}_0',\texttt{T}_1 \times ... \times \texttt{T}_\texttt{n}) = \{\texttt{methH}\} \\ \texttt{D},\Gamma \vdash_{se/e} \texttt{e}_\texttt{i} : \texttt{ET}_\texttt{i} \quad \texttt{i}\in\{\texttt{0...n}\}\end{array}}{\texttt{D},\Gamma \vdash_{se/e} \texttt{e}_0.[\texttt{T}_1 \times ... \times \texttt{T}_\texttt{n}]\texttt{m}(\texttt{e}_1...\texttt{e}_\texttt{n}) : (\cup\texttt{ET}_\texttt{i})_{\texttt{i=0}}^{\texttt{n}} \cup \mathit{ExcT}(\texttt{methH})}$$

$$\frac{\begin{array}{l}\texttt{D},\Gamma \vdash_{se/n} \texttt{e} : \texttt{E} \\ \texttt{D},\Gamma \vdash_{se/e} \texttt{e} : \texttt{ET}\end{array}}{\texttt{D},\Gamma \vdash_{se/e} \textbf{throw } \texttt{e} : \{\texttt{E}\}\cup\texttt{ET}}$$

$$\frac{\begin{array}{l}\texttt{n} \geq 0 \\ \texttt{D} \vdash \texttt{E}_\texttt{i} \sqsubseteq \texttt{Exception}, \quad \texttt{i}\in\{\texttt{1...n}\} \\ \texttt{D} \vdash \Gamma \diamond, \\ \Gamma(\texttt{v}_\texttt{i}) = \mathit{Undef} \quad \texttt{i}\in\{\texttt{1...n}\} \\ \texttt{D},\Gamma; \ \texttt{E}_\texttt{i} \ \texttt{v}_\texttt{i} \ \vdash_{se/e} \texttt{stmts}_\texttt{i} : \texttt{ET}_\texttt{i} \quad \texttt{i}\in\{\texttt{1...n}\} \\ \texttt{D},\Gamma \vdash_{se/e} \texttt{stmts}_0 : \texttt{ET}_0 \\ \texttt{D},\Gamma \vdash_{se/e} \texttt{stmts}_{\texttt{n+1}} : \texttt{ET}_{\texttt{n+1}}\end{array}}{\begin{array}{l}\texttt{D},\Gamma \vdash_{se/e} \textbf{try } \texttt{stmts}_0 \textbf{ catch } \texttt{E}_1 \ \texttt{v}_1 \ \texttt{stmts}_1 ... \textbf{ catch } \texttt{E}_\texttt{n} \ \texttt{v}_\texttt{n} \ \texttt{stmts}_\texttt{n} \ : \\ \qquad\qquad (\texttt{ET}_0 \setminus^{\texttt{e}}_{\texttt{D}} \{\texttt{E}_1, \ldots, \texttt{E}_\texttt{n}\}) \cup (\cup\texttt{ET}_\texttt{i})_{\texttt{i=1}}^{\texttt{n}} \\ \texttt{D},\Gamma \vdash_{se/e} \textbf{try } \texttt{stmts}_0 \textbf{ catch } \texttt{E}_1 \ \texttt{v}_1 \ \texttt{stmts}_1 ... \textbf{ catch } \texttt{E}_\texttt{n} \ \texttt{v}_\texttt{n} \ \texttt{stmts}_\texttt{n} \\ \qquad \textbf{finally } \ \texttt{stmts}_{\texttt{n+1}} \ : \ (\texttt{ET}_0 \setminus^{\texttt{e}}_{\texttt{D}} \{\texttt{E}_1, \ldots, \texttt{E}_\texttt{n}\}) \cup (\cup\texttt{ET}_\texttt{i})_{\texttt{i=1}}^{\texttt{n+1}}\end{array}}$$

Figure 22: *Abnormal types* for Java$_{se}$ terms

Definition 21 *For a Java$_{se}$ term* `t`*, program* `P`*, declaration* `D` *with* $\mathtt{D} = \mathcal{D}(\mathtt{P})$*, environment* Γ*, we define*

$$\begin{array}{lll}
\mathtt{P}, \Gamma \vdash_{se} \mathtt{t} \ : \ \tau & \textit{iff} & \mathtt{D}, \Gamma \vdash_{se} \mathtt{t} \ : \ \tau \\
\mathtt{P}, \Gamma \vdash_{se/n} \mathtt{t} : \mathtt{T} & \textit{iff} & \mathtt{D}, \Gamma \vdash_{se/n} \mathtt{t} : \mathtt{T}; \\
\mathtt{P}, \Gamma \vdash_{se/e} \mathtt{t} : \mathtt{ET} & \textit{iff} & \mathtt{D}, \Gamma \vdash_{se/e} \mathtt{t} : \mathtt{ET}
\end{array}$$

3.7 Java$_r$ – extended

Syntactic differences with Java$_{se}$

As we said in the previous section, Java$_{se}$ is an enriched version of Java$_s$, extended with compile-time type information necessary at run-time. Additionally, at run time, new terms may be reached, whose syntax is not covered by Java$_{se}$. Therefore we further extend Java$_{se}$, to obtain Java$_r$– the run time language. Java$_r$, defined in figure 23, is a pure superset of Java$_{se}$, because it describes additional artifacts that may arise at run-time only.

$$\begin{array}{lll}
\mathit{Stmts} & ::= & \epsilon \ \mid \mathit{Stmts}\ ;\ \mathit{Stmt} \\
\mathit{Stmt} & ::= & \mathbf{if}\ \mathit{Expr}\ \ \mathbf{then}\ \mathit{Stmts}\ \ \mathbf{else}\ \mathit{Stmts}\ \mid\ \mathit{Var} = \mathit{Expr} \\
 & \mid & \mathit{Expr}.[\mathit{ArgType}]\mathtt{MethName}\ ([\mathit{Expr}^*])\ \mid\ \mathbf{throw}\ \mathit{Expr} \\
 & \mid & \mathbf{try}\ \mathit{Stmts}\ (\mathbf{catch}\ \mathtt{ClassName}\ \mathtt{Id}\ \mathit{Stmts})^*\ \mathbf{finally}\ \mathit{Stmts} \\
 & \mid & \mathbf{try}\ \mathit{Stmts}\ (\mathbf{catch}\ \mathtt{ClassName}\ \mathtt{Id}\ \mathit{Stmts})^+ \\
\mathit{Expr} & ::= & \mathit{Value}\ \mid\ \mathit{Var}\ \mid \mathit{Expr}.[\mathit{ArgType}]\mathtt{MethName}\ (\mathit{Expr}^*) \\
 & \mid & \mathbf{new}\ \mathtt{[ArgType]ClassName}\ ([\mathit{Expr}^*])\ \mid\ \ \mathbf{new}\ \mathit{SimpleType}\ ([\mathit{Expr}])^+([])^* \\
 & \mid & \mathit{Stmts} \\
\mathit{Var} & ::= & \mathtt{Name}\ \mid\ \mathit{Expr}.[\mathtt{ClassName}]\mathtt{VarName}\ \mid\ \mathit{Var}[\mathit{Expr}]\ \mid\ \ \mathbf{this} \\
 & \mid & \iota_{\mathtt{i}}.[\mathtt{ClassName}]\mathtt{VarName} \mid \iota_{\mathtt{i}}[\mathit{Expr}] \qquad \mathtt{i}\ \textit{an integer} \\
 & \mid & \mathtt{null}.[\mathtt{ClassName}]\mathtt{VarName} \mid \mathtt{null}[\mathit{Expr}] \\
\mathit{Value} & ::= & \mathit{PrimValue}\ \mid\ \mathtt{null} \mid \mathit{RefValue} \\
\mathit{RefValue} & ::= & \iota_{\mathtt{i}} \qquad\qquad\qquad\qquad\qquad\qquad \mathtt{i}\ \textit{an integer} \\
\mathit{PrimValue} & ::= & \mathit{intValue}\ \mid \mathit{charValue}\ \mid \mathit{boolValue}\ \mid \ldots \\
\mathit{VarType} & ::= & \mathit{SimpleType}\ \mid \mathit{ArrayType} \\
\mathit{SimpleType} & ::= & \mathit{PrimType}\ \mid \mathtt{ClassName}\ \mid \mathtt{InterfaceName} \\
\mathit{ArrayType} & ::= & \mathit{SimpleType}\,[] \mid \mathit{ArrayType}\,[] \\
\mathit{PrimType} & ::= & \mathbf{bool}\ \ \mid \mathbf{char}\ \ \mid \mathbf{int}\ \ \mid \ldots
\end{array}$$

Figure 23: Java$_r$ terms

These additional artifacts, which may occur at run-time and are not part of Java$_{se}$, are part of Java$_r$, arising through addresses, the `null` value, and statements as expressions. Addresses have the form $\iota_{\mathtt{i}}$; they represent references to objects and arrays, and may appear wherever a value is expected, as well as in array and field accesses. Therefore, Java$_{se}$ variables may have the form $\iota_{\mathtt{i}}.[\mathtt{ClassName}]\mathtt{VarName}$, or $\iota_{\mathtt{i}}[\mathit{Expr}]$, and expressions may have the form $\iota_{\mathtt{i}}$. An access to `null` may arise during evaluation of array or field access variables, therefore Java$_r$ expressions may have the form $\mathtt{null}.[\mathtt{ClassName}]\mathtt{VarName}$, or $\mathtt{null}[\mathit{Expr}]$. Furthermore, in order to describe method evaluation through in-line expansion rather than closures and stacks, in Java$_r$ we allow an expression to consist of a sequence of statements, so that in the operational semantics a method call can be rewritten to a statement sequence.

3.7.1 Operational semantics – extended

Figure 24 describes the run-time model for the operational semantics. For a given program `P`, the operational semantics maps configurations to new configurations. Configurations are tuples of Java$_r$ terms and states, or just states. The operational semantics is a mapping from programs and configurations to configurations.

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

An object is annotated by its class, and it consists of a sequence of labels and values. Each label also carries the class in which it was defined; this is needed for labels shadowing labels from superclasses, *c.f.* [8] ch. 9.5. For the animals example, $\ll$ `isHealthy Animal: true`, `numLives Cat: 7` $\gg^{\texttt{Cat}}$ is an object of class `Cat`. It inherits the field `isHealthy` from `Animal`, and has the field `numLives` from `Cat`. A class and interface entry consists of field names and values for those fields. All the fields will be **static** fields defined in that class or interface.

The following state σ_0 contains mappings appropriate for the animals example:

$$
\begin{array}{lcl}
\sigma_0(\texttt{Animal}) & = & \iota_{\texttt{h}} \\
\sigma_0(\texttt{Cat}) & = & \iota_{\texttt{i}} \\
\sigma_0(\texttt{Dog}) & = & \iota_{\texttt{j}} \\
\sigma_0(\texttt{Carnivore}) & = & \iota_{\texttt{k}} \\
 & & \\
\sigma_0(\iota_{\texttt{h}}) & = & \ll\ \gg \\
\sigma_0(\iota_{\texttt{i}}) & = & \ll \texttt{numCatsAlive:}\ \ 5 \gg \\
\sigma_0(\iota_{\texttt{j}}) & = & \ll\ \gg \\
\sigma_0(\iota_{\texttt{k}}) & = & \ll \texttt{diet:}\ \ \iota_{\texttt{m}} \gg \\
 & & \\
\sigma_0(\iota_{\texttt{m}}) & = & \ll \texttt{isHealthy Animal:}\ \ \texttt{false}\ ,\texttt{numLives Cat:}\ \ 2 \gg^{\texttt{Cat}}
\end{array}
$$

Arrays carry their dimension and type information, and they consist of a sequence of values for the first dimension. For example, $[\![\texttt{3},\texttt{5},\texttt{8},\texttt{11}]\!]^{\texttt{int[]}}$ is a one dimensional array of integers.

$$
\begin{array}{lcl}
\textit{Configuration} & ::= & \text{Java}_r \text{ term}, \textit{state} \cup \textit{state} \\
\leadsto & : & \text{Java}_{se} \text{ program} \longrightarrow \textit{Configuration} \longrightarrow \textit{Configuration} \\
\underset{\texttt{P}}{\leadsto} & : & \textit{Configuration} \longrightarrow \textit{Configuration} \\
\textit{State} & ::= & (\texttt{Ident} \longrightarrow \textit{Value}\ )^* \cup \\
 & & (\ \textit{RefValue} \longrightarrow \textit{ObjectOrArray}\ )^* \\
\textit{ObjectOrArray} & ::= & \textit{Object}\ \mid\ \textit{Array}\ \mid\ \textit{ClassOrInterface} \\
\textit{Object} & ::= & \ll(\texttt{LabelName ClassName} : \textit{Value}\ )^* \gg^{\texttt{ClassName}} \\
\textit{Array} & ::= & [\![(\textit{Value})^*]\!]^{\textit{ArrayType}} \\
\textit{ClassOrInterface} & ::= & \ll(\texttt{LabelName} : \textit{Value}\ )^*\ \gg
\end{array}
$$

Figure 24: Java$_r$ run-time model

3.7.2 State operations

We now define operations on objects, classes, interfaces, 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 and the class or interface are well-defined.

Definition 22 *For a class or interface* $X = \ll \texttt{f}_1\ \texttt{C}_1 : \texttt{val}_1, \texttt{l}_2\ \texttt{C}_2 : \texttt{val}_2, \ldots \texttt{f}_\texttt{n}\ \texttt{C}_\texttt{n} : \texttt{val}_\texttt{n} \gg$*, state* σ*, reference* $\iota_{\texttt{i}}$ *and a field identifier* `f`*, we define:*

- *the* access to component $\mathtt{f}$ of a class or interface *stored at reference* $\iota_{\mathtt{i}}$, *in state* σ :

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

Definition 23 *For object* $\mathtt{obj} = \ll \mathtt{f}_1\ \mathtt{C}_1 : \mathtt{val}_1,\ \mathtt{l}_2\ \mathtt{C}_2 : \mathtt{val}_2,\ ...,\ \mathtt{f}_\mathtt{n}\ \mathtt{C}_\mathtt{n} : \mathtt{val}_\mathtt{n} \gg^{\mathtt{C}'}$, *state* σ, *value* $\mathtt{val}$, *reference* $\iota_{\mathtt{i}}$, *identifier or reference* $\mathtt{z}$, *class* $\mathtt{C}$, *field identifier* $\mathtt{f}$, *integers* $\mathtt{m}$, $\mathtt{k}$ *with* $\mathtt{m} \geq 0$, *array* $\mathtt{arr} = [\![\mathtt{val}_0, ...\mathtt{val}_{\mathtt{n}-1}]\!]^{\mathtt{T}[]_1 ... []_\mathtt{m}}$, *we define:*

- *the* access to field $\mathtt{f}$ declared in class $\mathtt{C}$:

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

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

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

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

  $$\mathtt{arr}[\mathtt{k}] \quad = \quad \mathtt{val}_\mathtt{k} \quad \text{if } 0 \leq \mathtt{k} \leq \mathtt{n} - 1$$

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

  $$\begin{array}{lcll} \sigma'(\mathtt{z}) & = & \mathtt{val} & \\ \sigma'(\mathtt{z}') & = & \sigma(\mathtt{z}') & \text{for } \mathtt{z}' \neq \mathtt{z} : \end{array}$$

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

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

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

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

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

Definition 24 *A* $Java_r$ *term* $\mathtt{t}$ *is*

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

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

3.7.3 Program execution

Figures 25, 26, 27 and 30 describe the rewriting of $Java_r$ terms. We chose small step semantics because we found this more intuitive. Interestingly, it turns out that large step semantics allow for a simpler proof of subject reduction, and in particular, do not require different type rules for $Java_r$ assignment to array components and the other assignments statements. On the other hand, only small step semantics allows the description of co-routines.

The rules for program execution require properties deduced from a program. We "redefine" properties needed for the operational semantics, which were defined for declarations in section 3.6, and also use definitions 11 and 12 from section 3.3.11.

Definition 25 *For a $Java_{se}$ program* $\mathtt{P}$*, declaration* $\mathtt{D} = \mathcal{D}(\mathtt{P})$*, identifiers* $\mathtt{C}$*,* $\mathtt{C}'$*,* $\mathtt{I}$*,* $\mathtt{I}'$*,* $\mathtt{f}$*,* $\mathtt{m}$*, class or interface type* $\mathtt{T}$ *and argument type* $\mathtt{AT}$

$$
\begin{array}{l}
\mathtt{P} \vdash \mathtt{C} \sqsubseteq \mathtt{C}' \quad \textit{iff} \quad \mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C}' \\
\mathtt{P} \vdash \mathtt{I} \leq \mathtt{I}' \quad \textit{iff} \quad \mathtt{D} \vdash \mathtt{I} \leq \mathtt{I}' \\
\mathit{Fields}(\mathtt{P}, \mathtt{C}, \mathtt{f}) = \mathit{Fields}(\mathtt{D}, \mathtt{C}, \mathtt{f}) \\
\mathit{FirstFit}(\mathtt{P}, \mathtt{m}, \mathtt{T}, \mathtt{AT}) = \{\mathtt{meth} \mid \mathtt{meth} \in \mathit{Meths}(\mathtt{P}, \mathtt{T}, \mathtt{m}) \ \ \textit{and} \ \ \mathit{ArgT}(\mathtt{meth}) = \mathtt{AT}\} \\
\mathit{FirstFitConstr}(\mathtt{P}, \mathtt{C}, \mathtt{AT}) = \{\mathtt{constr} \mid \mathtt{constr} \in \mathit{ConstrDef}(\mathtt{P}, \mathtt{C}) \ \ \textit{and} \ \ \mathit{ArgT}(\mathtt{constr}) = \mathtt{AT}\}
\end{array}
$$

Note, that $\mathit{FirstFit}(\mathtt{P}, \mathtt{m}, \mathtt{T}, \mathtt{AT})$ returns the whole method including the method body, whereas $\mathit{FirstFit}(\mathtt{D}, \mathtt{m}, \mathtt{T}, \mathtt{AT})$ returns a method header only.

$$
\frac{}{\mathtt{id}, \sigma \underset{P}{\leadsto} \sigma(\mathtt{id}), \sigma}
$$

$$
\frac{\mathtt{v}, \sigma \underset{P}{\leadsto} \mathtt{v}', \sigma'}{\begin{array}{l} \mathtt{v}[\mathtt{e}], \sigma \underset{P}{\leadsto} \mathtt{v}'[\mathtt{e}], \sigma' \\ \mathtt{v}.[\mathtt{C}]\mathtt{f}, \sigma \underset{P}{\leadsto} \mathtt{v}'.[\mathtt{C}]\mathtt{f}, \sigma' \end{array}}
$$

$$
\frac{\mathtt{k} \text{ is integer value}}{\iota_{\mathtt{i}}[\mathtt{k}], \sigma \underset{P}{\leadsto} \sigma(\iota_{\mathtt{i}})[\mathtt{k}], \sigma}
$$

$$
\frac{\begin{array}{l}
\iota_{\mathtt{i}} \text{ is new in } \sigma \\
\forall \mathtt{f}, \mathtt{C}', \mathtt{T}' \text{ with } (\mathtt{fMod}\ \mathtt{C}'\ \mathtt{T}'\ \mathtt{f}) \in \mathit{Fields}(\mathtt{P}, \mathtt{C}, \mathtt{f}) \\
\quad \text{and } \ \textbf{static} \notin \mathtt{fMod}: \\
\qquad \exists_1 \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n}\} : \mathtt{f}_{\mathtt{i}} = \mathtt{f}, \mathtt{C}_{\mathtt{i}} = \mathtt{C}', \mathtt{T}_{\mathtt{i}} = \mathtt{T}' \\
\mathtt{v}_{\mathtt{i}} = \mathit{InitVal}(\mathtt{P}, \mathtt{T}_i), \qquad \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{n}\} \\
\sigma' = \sigma[\iota_{\mathtt{i}} \mapsto \ll \mathtt{f}_1\ \mathtt{C}_1 : \mathtt{v}_1, \ldots \mathtt{f}_{\mathtt{n}}\ \mathtt{C}_{\mathtt{n}} : \mathtt{v}_{\mathtt{n}} \gg^{\mathtt{C}}] \\
\sigma'' = \sigma'[\textbf{this} \mapsto \iota_{\mathtt{j}}] \\
[\mathtt{T}_1 \times \ldots \times \mathtt{T}_{\mathtt{n}}]\mathtt{C}(\mathtt{e}_1 \ldots \mathtt{e}_{\mathtt{n}}), \sigma'' \underset{P}{\leadsto} \iota_{\mathtt{i}}, \sigma''' \\
\sigma'''' = \sigma'''[\textbf{this} \mapsto \sigma(\mathtt{this})]
\end{array}}{\textbf{new} \ \ [\mathtt{T}_1 \times \ldots \times \mathtt{T}_{\mathtt{n}}]\mathtt{C}(\mathtt{e}_1 \ldots \mathtt{e}_{\mathtt{n}}), \sigma \underset{P}{\leadsto} \iota_{\mathtt{i}}, \sigma''''}
$$

$$
\frac{\begin{array}{l}
\mathtt{1} \leq \mathtt{j} \leq \mathtt{k}, \quad \mathtt{k} \geq 1, \quad \mathtt{m} \geq 0 \\
\mathtt{n}_{\mathtt{i}} \geq 0 \qquad \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{j} - \mathtt{1}\} \\
\mathtt{n}_{\mathtt{j}}, \sigma \underset{P}{\leadsto} \mathtt{n}'_{\mathtt{j}}, \sigma'
\end{array}}{\begin{array}{l}
\textbf{new } \mathtt{T}[\mathtt{n}_1] \ldots [\mathtt{n}_{\mathtt{j}}] \ldots [\mathtt{n}_{\mathtt{k}}][]_1 \ldots []_{\mathtt{m}}, \sigma \\
\underset{P}{\leadsto} \textbf{new } \mathtt{T}[\mathtt{n}_1] \ldots [\mathtt{n}'_{\mathtt{j}}] \ldots [\mathtt{n}_{\mathtt{k}}][]_1 \ldots []_{\mathtt{m}}, \sigma'
\end{array}}
$$

$$
\frac{\begin{array}{l}
\mathtt{k} \geq 1, \mathtt{m} \geq 0 \\
\mathtt{n}_{\mathtt{i}} \text{ ground} \quad \mathtt{i} \in \{\mathtt{1} \ldots \mathtt{k}\} \\
\mathtt{n}_{\mathtt{j}} < 0 \text{ for some} \quad \mathtt{j} \in \{\mathtt{1} \ldots \mathtt{k}\} \\
\textbf{new} \ \ \mathtt{NegSzeE}, \sigma \underset{P}{\leadsto} \iota_{\mathtt{i}}, \sigma'
\end{array}}{\textbf{new } \mathtt{T}[\mathtt{n}_1] \ldots [\mathtt{n}_{\mathtt{k}}][]_1 .. []_{\mathtt{m}}, \sigma \underset{P}{\leadsto} \textbf{throw} \ \ \iota_{\mathtt{i}}, \sigma'}
$$

$$
\frac{}{\iota_{\mathtt{i}}.[\mathtt{C}]\mathtt{f}, \sigma \underset{P}{\leadsto} \sigma(\iota_{\mathtt{i}}, \mathtt{f}, \mathtt{C}), \sigma}
$$

$$
\frac{\mathtt{e}, \sigma \underset{P}{\leadsto} \mathtt{e}', \sigma'}{\begin{array}{l} \iota_{\mathtt{i}}[\mathtt{e}], \sigma \underset{P}{\leadsto} \iota_{\mathtt{i}}[\mathtt{e}'], \sigma' \\ \mathtt{null}[\mathtt{e}], \sigma \underset{P}{\leadsto} \mathtt{null}[\mathtt{e}'], \sigma' \end{array}}
$$

$$
\frac{\begin{array}{l} \mathtt{k} \text{ is integer value} \\ \textbf{new} \ \ \mathtt{NullPE}, \sigma \underset{P}{\leadsto} \iota_{\mathtt{i}}, \sigma' \end{array}}{\begin{array}{l} \mathtt{null}[\mathtt{k}], \sigma \underset{P}{\leadsto} \textbf{throw} \ \ \iota_{\mathtt{i}}, \sigma' \\ \mathtt{null}.[\mathtt{C}]\mathtt{f}, \sigma \underset{P}{\leadsto} \textbf{throw} \ \ \iota_{\mathtt{i}}, \sigma' \end{array}}
$$

$$
\frac{\begin{array}{l}
\iota_{\mathtt{i}} \text{ is new in } \sigma \\
\mathtt{T} \text{ is a simple type} \\
\mathtt{v} = \mathit{InitVal}(\mathtt{P}, \mathtt{T}) \\
\mathtt{v} = \mathtt{v}_0 = \mathtt{v}_1 \ldots = \mathtt{v}_{\mathtt{n}-1}, \quad \mathtt{n} \geq 0 \\
\sigma' = \sigma[\iota_{\mathtt{i}} \mapsto [\![\mathtt{v}_0, \ldots \mathtt{v}_{\mathtt{n}-1}]\!]^{\mathtt{T}[]}]
\end{array}}{\textbf{new} \ \ \mathtt{T}[\mathtt{n}], \sigma \underset{P}{\leadsto} \iota_{\mathtt{i}}, \sigma'}
$$

$$
\frac{\begin{array}{l}
\mathtt{m} \geq 1, \mathtt{n} \geq 0, k \geq 2 \\
\iota_{\mathtt{i}} \text{ new in } \sigma \\
\sigma' = \sigma[\iota_{\mathtt{i}} \mapsto [\![\mathtt{null}_0, \ldots \mathtt{null}_{\mathtt{n}-1}]\!]^{\mathtt{T}[]_1 \ldots []_{\mathtt{k}}}]
\end{array}}{\textbf{new } \mathtt{T}[\mathtt{n}][]_2 \ldots []_{\mathtt{k}}, \sigma \underset{P}{\leadsto} \iota_{\mathtt{i}}, \sigma'}
$$

$$
\frac{\begin{array}{l}
\mathtt{n}_1 \geq 0, \ \mathtt{k} \geq 2, \ \mathtt{m} \geq 0, \ \sigma_0 = \sigma \\
\mathtt{T} \text{ is a simple type} \\
\textbf{new} \ \ \mathtt{T}[\mathtt{n}_2] \ldots [\mathtt{n}_{\mathtt{k}}][]_1 \ldots []_{\mathtt{m}}, \sigma_i \underset{P}{\leadsto} \iota_{\mathtt{j}_{\mathtt{i}}}, \sigma_{i+1} \\
\qquad \text{for all} \ \ \mathtt{i} \in \{\mathtt{0} \ldots \mathtt{n}_1 - \mathtt{1}\} \\
\iota_{\mathtt{j}_{\mathtt{i}}} \text{ is new in } \sigma_{\mathtt{n}_{\mathtt{i}}} \qquad \mathtt{i} \in \{\mathtt{0} \ldots \mathtt{n}_1\} \\
\sigma' = \sigma_{n_1}[\iota_{\mathtt{j}_{n_1}} \mapsto [\![\iota_{\mathtt{j}_0}, \ldots, \iota_{\mathtt{j}_{\mathtt{n}_1 - 1}}]\!]^{\mathtt{T}[]_1 \ldots []_{\mathtt{k}+\mathtt{m}}}]
\end{array}}{\textbf{new} \ \ \mathtt{T}[\mathtt{n}_1] \ldots [\mathtt{n}_{\mathtt{k}}][]_1 \ldots []_{\mathtt{m}}, \sigma \underset{P}{\leadsto} \iota_{\mathtt{j}_{\mathtt{n}_1}}, \sigma'}
$$

Figure 25: Expression execution

In figure 25 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 27 prevent an expression like x, or $\iota_{\mathtt{i}}.[\mathtt{C}]\mathtt{v}$, appearing on the left hand side of an assignment from being rewritten further. They allow an expression of the form w.[C1]x.[C2]y.[C3]z to be rewritten to an expression of the form $\iota_{\mathtt{j}}.[\mathtt{C}]\mathtt{v}$ for some j. Furthermore, there is *no* rule of the form $\iota_{\mathtt{j}}, \sigma \rightsquigarrow_{\mathtt{P}} \sigma(\iota_{\mathtt{j}}), \sigma$. This is because there is no explicit dereferencing operator in Java. Objects are passed as references, and they are dereferenced only implicitly, when their fields are accessed.

Array access as described here adheres to the rules in ch. 15.12 of [8], 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 25 describe the creation of new objects or arrays, *c.f.* ch. 15.8-15.9 of [8]. 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 [8] and here in the following definition.

Definition 26 *The* initial value *for a simple type* T *defined in a program* P *is* :

$$
InitVal(\mathtt{P},\mathtt{T}) = \begin{cases} \mathtt{0} & \textit{iff} \quad \mathtt{T} = \textbf{int} \\ \mathtt{'\ '} & \textit{iff} \quad \mathtt{T} = \textbf{char} \\ \mathtt{false} & \textit{iff} \quad \mathtt{T} = \textbf{bool} \\ \mathtt{null} & \textit{iff} \quad \mathtt{P} \vdash \mathtt{T} \sqsubseteq \mathtt{T} \ \ \textit{or} \ \ \mathtt{P} \vdash \mathtt{T} \leq \mathtt{T} \end{cases}
$$

For example, for a state σ_{00} the expression **new int**[2][3][][] would be executed as:
new int[2][3][][], $\sigma_{00} \rightsquigarrow_{\mathtt{P}} \iota_7, \sigma_{01}$
where ι_5, ι_6, and ι_7 are new in σ_{00}, and they have the following contents in σ_{01}:

$$
\begin{aligned}
\sigma_{01}(\iota_5) &= [\![\mathtt{null},\mathtt{null},\mathtt{null}]\!]^{\mathtt{int}[][][]} \\
\sigma_{01}(\iota_6) &= [\![\mathtt{null},\mathtt{null},\mathtt{null}]\!]^{\mathtt{int}[][][]} \\
\sigma_{01}(\iota_7) &= [\![\iota_5,\iota_6]\!]^{\mathtt{int}[][][][]}
\end{aligned}
$$

$$\frac{\mathtt{stmts}, \sigma \rightsquigarrow_{\mathtt{P}} \sigma'}{\mathtt{stmts};\mathtt{stmt}, \sigma \rightsquigarrow_{\mathtt{P}} \mathtt{stmt}, \sigma'} \qquad \frac{\mathtt{stmts}, \sigma \rightsquigarrow_{\mathtt{P}} \mathtt{stmts}', \sigma'}{\mathtt{stmts};\mathtt{stmt}, \sigma \rightsquigarrow_{\mathtt{P}} \mathtt{stmts}';\mathtt{stmt}, \sigma'}$$

$$\frac{}{\textbf{if true then } \mathtt{stmts} \textbf{ else } \mathtt{stmts}', \sigma \rightsquigarrow_{\mathtt{P}} \mathtt{stmts}, \sigma} \qquad \frac{\mathtt{e}, \sigma \rightsquigarrow_{\mathtt{P}} \mathtt{e}', \sigma'}{\textbf{if } \mathtt{e} \textbf{ then } \mathtt{stmts} \textbf{ else } \mathtt{stmts}', \sigma \rightsquigarrow_{\mathtt{P}} \textbf{if } \mathtt{e}' \textbf{ then } \mathtt{stmts} \textbf{ else } \mathtt{stmts}', \sigma'}$$

$$\frac{}{\textbf{if false then } \mathtt{stmts} \textbf{ else } \mathtt{stmts}', \sigma \rightsquigarrow_{\mathtt{P}} \mathtt{stmts}', \sigma} \qquad \frac{}{\textbf{return }, \sigma \rightsquigarrow_{\mathtt{P}} \sigma}$$

$$\frac{\mathtt{e}, \sigma \rightsquigarrow_{\mathtt{P}} \mathtt{e}', \sigma'}{\textbf{return } \mathtt{e}, \sigma \rightsquigarrow_{\mathtt{P}} \textbf{return } \mathtt{e}', \sigma'} \qquad \frac{\mathtt{val} \text{ is ground}}{\textbf{return } \mathtt{val}, \sigma \rightsquigarrow_{\mathtt{P}} \mathtt{val}, \sigma}$$

Figure 26: Statement Execution

Figure 26 describes statement execution. Statement sequences are evaluated from left to right. In conditional statements the condition is evaluated first; if it evaluates to true, then the first branch is executed, otherwise the second branch is executed. A **return** statement terminates execution. A statement returning an expression evaluates this expression until ground and replaces itself by this ground value – thus modeling methods returning values.

$$\frac{\begin{array}{l}\mathtt{v}\text{ is not l-ground}\\ \mathtt{v},\sigma\leadsto_{\mathtt{P}}\mathtt{v}',\sigma'\end{array}}{\mathtt{v}{=}\mathtt{e},\sigma\leadsto_{\mathtt{P}}\mathtt{v}'{=}\mathtt{e},\sigma'}\qquad\frac{\begin{array}{l}\mathtt{v}\text{ is l-ground}\\ \mathtt{e},\sigma\leadsto_{\mathtt{P}}\mathtt{e}',\sigma'\end{array}}{\mathtt{v}{=}\mathtt{e},\sigma\leadsto_{\mathtt{P}}\mathtt{v}{=}\mathtt{e}',\sigma'}$$

$$\frac{\begin{array}{l}\mathtt{val}\text{ is ground}\\ \mathtt{id}\text{ is an identifier}\end{array}}{\begin{array}{l}\mathtt{id}{=}\mathtt{val},\sigma\leadsto_{\mathtt{P}}\sigma[\mathtt{id}\mapsto\mathtt{val}]\\ \iota_{\mathtt{i}}.[\mathtt{C}]\mathtt{v}{=}\mathtt{val},\sigma\\ \qquad\leadsto_{\mathtt{P}}\sigma[\iota_{\mathtt{i}},\mathtt{v},\mathtt{C}\mapsto\mathtt{val}]\end{array}}\qquad\frac{\begin{array}{l}\mathtt{val},\mathtt{k}\text{ are ground}\\ \sigma(\iota_{\mathtt{i}})=[\![\mathtt{val}_0...\mathtt{val}_{n-1}]\!]^{\mathtt{T}[]_1...[]_m}\\ 0>\mathtt{k},\ \text{ or }\mathtt{k}>n-1\\ \mathbf{new}\ \mathtt{IndOutBndE},\sigma\leadsto_{\mathtt{P}}\iota_{\mathtt{j}},\sigma'\end{array}}{\iota_{\mathtt{i}}[\mathtt{k}]{=}\mathtt{val},\sigma\leadsto_{\mathtt{P}}\mathbf{throw}\ \ \iota_{\mathtt{j}},\sigma'}$$

$$\frac{\begin{array}{l}\mathtt{val},\mathtt{k}\text{ are ground}\\ \sigma(\iota_{\mathtt{i}})=[\![\mathtt{val}_0...\mathtt{val}_{n-1}]\!]^{\mathtt{T}[]_1...[]_m}\\ 0\leq\mathtt{k}\leq n-1\\ \mathtt{val}\text{ does not fit }\mathtt{T}[]_1...[]_m\text{ in }\mathtt{p},\sigma\\ \mathbf{new}\ \mathtt{ArrStoreE},\sigma\leadsto_{\mathtt{P}}\iota_{\mathtt{j}},\sigma'\end{array}}{\iota_{\mathtt{i}}[\mathtt{k}]{=}\mathtt{val},\sigma\leadsto_{\mathtt{P}}\mathbf{throw}\ \ \iota_{\mathtt{j}},\sigma'}\qquad\frac{\begin{array}{l}\mathtt{val},\mathtt{k}\text{ are ground}\\ \sigma(\iota_{\mathtt{i}})=[\![\mathtt{val}_0...\mathtt{val}_{n-1}]\!]^{\mathtt{T}[]_1...[]_m}\\ 0\leq\mathtt{k}\leq n-1\\ \mathtt{val}\text{ fits }\mathtt{T}[]_1...[]_m\text{ in }\mathtt{P},\sigma\end{array}}{\iota_{\mathtt{i}}[\mathtt{k}]{=}\mathtt{val},\sigma\leadsto_{\mathtt{P}}\sigma[\iota_{\mathtt{i}},\mathtt{k}\mapsto\mathtt{val}]}$$

Figure 27: Assignment execution

Figure 27 describes the evaluation of assignments. According to the first rule, the left hand side is evaluated first, until it becomes l-ground. Then, according to the next rule, the right hand side of the assignment is evaluated, up to the point of obtaining a ground term. Assignment to variables or to object components modifies the state accordingly.

The last three rules describe assignment to array components where the index being within bounds has to be checked first (if not, `IndOutBndE` is thrown), then the value has to fit the array (if not, `ArrStoreE` is thrown), and, if the two above requirements are satisfied, then the assignment is performed. Fitting, a requirement which ensures that an object or array value is of a type that can be appropriately stored into another array, is described in the definition 27.

Other exceptions (*e.g.* `null` access) need not be considered in these rules, because they are checked by the variable rules (figure 26), and then propagated by the exception rules from figure 30. Also, we have *no* rule of the form $\iota_{\mathtt{j}} := \mathtt{value},\sigma\leadsto_{\mathtt{P}}...$. This is because in Java overwriting of objects is not possible – only sending messages to them, or overwriting selected instance variables.

Definition 27 *A value* `val` fits *a type* $\mathtt{T}=\mathtt{T}'[]$ *in a program* `P`*, iff* `val` *is primitive, or* $\mathtt{val}{=}\mathtt{null}$*, or* $\sigma(\mathtt{val})=\ll...\gg^{\mathtt{C}}$ *and* $\mathtt{P}\vdash\mathtt{C}\sqsubseteq\mathtt{T}'$*, or* $\sigma(\mathtt{val})=[\![...]\!]^{\mathtt{T}''}$ *and* $\mathtt{P}\vdash\mathtt{T}''\sqsubseteq\mathtt{T}'$*.*

Note that a primitive value fits any array type. This is so, because when primitive values are assigned to array components no run time check needs to be performed. Also, note that in the above definition the types $\mathtt{T}'$ and $\mathtt{T}''$ may be array types themselves, and remember that the subclass relationship is monotonic with the array type constructor (*i.e.* $\mathtt{P}\vdash\mathtt{C}\sqsubseteq\mathtt{C}'$ implies that $\mathtt{P}\vdash\mathtt{C}[]\sqsubseteq\mathtt{C}'[]$).

Figure 28 describes the evaluation of method calls. The receiver and argument expressions are evaluated left to right. The first rule describes rewriting the $\mathtt{k}^{th}$ expression, where all the previous expressions (*i.e.* $\mathtt{val}_{\mathtt{i}}, \mathtt{i}\in\{\mathtt{1}...\mathtt{k}-\mathtt{1}\}$) are ground. The second rule requires the exception `NullPE` to be thrown if the receiver is `null`. The third rule describes dynamic method look up, taking into account the argument types, and the statically calculated method descriptor `AT`. The term $\mathtt{t}[\mathtt{t}'/\mathtt{x}]$ has the usual meaning of replacing the variable `x` by the term $\mathtt{t}'$ in the term `t`.

Figure 29 describes the evaluation of explicit invocations of constructors. The argument expressions are evaluated from left to right. The first rule describes how the `k` expressions are rewritten until they are all ground values. The second rule describes dynamic constructor look up, as determined by the argument type calculated at compile-time. Again, the term $\mathtt{t}[\mathtt{t}'/\mathtt{x}]$ has the usual meaning of replacing the variable `x` by the term $\mathtt{t}'$ in the term `t`. Another important point

$$\frac{\begin{array}{l}\mathtt{val_i}\ \text{ is ground for } \mathtt{i} \in \{\mathtt{0...k-1}\}, \mathtt{n} \geq \mathtt{k} \geq 0 \\ \mathtt{e_k}, \sigma \underset{\mathtt{P}}{\leadsto} \mathtt{e'_k}, \sigma'\end{array}}{\begin{array}{l}\mathtt{val_0.[AT]m(val_1, ..., val_{k-1}, e_k, ...e_n)}, \sigma \underset{\mathtt{P}}{\leadsto} \\ \mathtt{val_0.[AT]m(val_1, ..., val_{k-1}, e'_k, ...e_n)}, \sigma'\end{array}}$$

$$\frac{\mathtt{val_i}\quad \text{is ground for } \mathtt{i} \in \{\mathtt{1...n}\}, \mathtt{n} \geq 1}{\mathtt{null.[AT]m(val_1, ...val_n)}, \sigma \underset{\mathtt{P}}{\leadsto} \mathbf{throw\ \ new}\ \ \mathtt{NullPE}, \sigma}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0 \\ \mathtt{val_i}\quad \text{is ground for } \mathtt{i} \in \{\mathtt{0...n}\} \\ \sigma(\mathtt{val_0}) = \ll ... \gg^{\mathtt{C}} \\ \mathit{FirstFit}(\mathtt{P, m, C, AT}) = \mathtt{meth} \\ \mathtt{meth} = \mathtt{T\ m(T_1\ p_1,...,T}_n\ \mathtt{p}_n\mathtt{)}\ \mathbf{throws}\ \mathtt{E_1,...,E}_s\ \{\mathtt{mBody}\} \\ \mathtt{z_i} \text{ are new identifiers in } \sigma \\ \sigma' = \sigma[\mathtt{z_1} \mapsto \mathtt{val_1}]...[\mathtt{z_n} \mapsto \mathtt{val_n}] \\ \mathtt{mBody'} = \mathtt{mBody[z_0/this, z_1/p_1, ...z_n/p_n]}\end{array}}{\mathtt{val_0.[AT]m(val_1, ..., val_n)}, \sigma \underset{\mathtt{P}}{\leadsto} \mathtt{mBody'}, \sigma'}$$

Figure 28: Evaluation of method call

of this rule is the explicit constructor invocation statement that is the first execution step in the *current* constructor. If this involves calling a constructor of the parent class, initialisation of the instance variables declared in the *current* class is carried out right after the call to the parent class constructor returns – this is represented by the statement block $\mathtt{I_{Expr}}$. These are duly followed by the regular statement body comprising the constructor.

Definition 28 *For a Java program* P *and a class* C *declared in* P, InitExpr(P, C) *returns all the initialisation expressions for the instance variables of class C.*

$$\begin{array}{rl}\mathtt{InitExpr(P, C)} = \{\ (\mathtt{f = init_{expr};}) & |\quad (\mathtt{fMod\ T\ f\ init_{expr}}) \in \mathit{FSpec}(\mathtt{D, C, f}) \\ & \quad \mathit{and}\ \ \mathbf{static} \notin \mathtt{fMod}\ \ \mathit{and}\ \ \mathtt{init_{expr}} \neq \epsilon\}.\end{array}$$

$$\frac{\begin{array}{l}\mathtt{val_i}\ \text{ is ground for } \mathtt{i} \in \{\mathtt{0...k-1}\}, \mathtt{n} \geq \mathtt{k} \geq 0 \\ \mathtt{e_k}, \sigma \underset{\mathtt{P}}{\leadsto} \mathtt{e'_k}, \sigma'\end{array}}{\begin{array}{l}\mathtt{[AT]C(val_1, ..., val_{k-1}, e_k, ..., e_n)}, \sigma \underset{\mathtt{P}}{\leadsto} \\ \quad \mathtt{[AT]C(val_1, ..., val_{k-1}, e'_k, ..., e_n)}, \sigma'\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0 \\ \mathtt{val_i}\quad \text{is ground for } \mathtt{i} \in \{\mathtt{0...n}\} \\ \mathit{FirstFitConstr}(\mathtt{P, C, AT}) = \mathtt{constr} \\ \mathtt{constr} = \mathtt{cMod\ C(T_1\ p_1,...,T}_n\ \mathtt{p}_n\mathtt{)}\ \mathbf{throws}\ \mathtt{E_1,...,E}_s\ \{\mathtt{constrBody}\} \\ \mathtt{constrBody} = \mathtt{ExplicitConstrCall; constrRest} \\ \mathtt{z_i} \text{ are new identifiers in } \sigma \\ \sigma' = \sigma[\mathtt{z_1} \mapsto \mathtt{val_1}]...[\mathtt{z_n} \mapsto \mathtt{val_n}] \\ \mathit{exp} = \mathtt{ExplicitConstrInvocation[C/}\mathbf{this}, \mathit{SuperC}(\mathtt{P, C})/\mathbf{super}] \\ \mathbf{this} \in \mathtt{ExplicitConstrInvocation}\ \text{ and }\ \mathtt{I_{Expr}} = \epsilon \\ \quad \text{or }\ \mathbf{super} \in \mathtt{ExplicitConstrInvocation}\ \text{ and }\ \mathtt{I_{Expr}} = \mathtt{InitExpr(P, C)} \\ \mathtt{crr} = \mathtt{constrRest[z_0/this, z_1/p_1, ...z_n/p_n]}\end{array}}{\mathtt{[AT]C(val_1, ..., val_n)}, \sigma \underset{\mathtt{P}}{\leadsto} \mathtt{(exp; I_{Expr}; crr)}, \sigma'}$$

Figure 29: Evaluation of an explicit constructor invocation

The rules in figure 30 describe the operational semantics for propagation and handling exceptions. Thus, **try** ... **throw new** $\mathtt{E}$**;** ...**catch** $\mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1},\sigma$ would rewrite to **try** ... **throw** $\iota_{\mathtt{i}}$**; catch** $\mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1},\sigma$, and then, if $\mathtt{E}$ is a subclass of $\mathtt{E_1}$, to $\mathtt{stmts}'_1,\sigma$.

$$\frac{\begin{array}{l}\mathtt{e},\sigma \rightsquigarrow_{\mathtt{P}} \mathtt{e}',\sigma' \\ \mathtt{cont}@\cdot\mathrm{A} \text{ a context}\end{array}}{\begin{array}{l}\mathtt{cont}@\mathbf{throw}\ \mathtt{e}\mathrm{A},\sigma \rightsquigarrow_{\mathtt{P}} \mathtt{cont}@\mathbf{throw}\ \mathtt{e}'\mathrm{A},\sigma' \\ \mathtt{cont}@\mathbf{throw}\ \iota_{\mathtt{i}}\mathrm{A},\sigma \rightsquigarrow_{\mathtt{P}} \mathbf{throw}\,\iota_{\mathtt{i}},\sigma\end{array}}
\qquad
\frac{}{\begin{array}{l}\mathbf{throw}\ \mathtt{null},\sigma \\ \quad\rightsquigarrow_{\mathtt{P}} \mathbf{throw}\ \mathbf{new}\ \mathtt{NullPE},\sigma\end{array}}$$

$$\frac{\mathtt{stmts},\sigma \rightsquigarrow_{\mathtt{P}} \sigma'}{\begin{array}{l}\mathbf{try}\ \mathtt{stmts}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ \ldots\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n},\sigma \rightsquigarrow_{\mathtt{P}} \sigma' \\ \mathbf{try}\ \mathtt{stmts}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ \ldots\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ \mathbf{finally}\ \mathtt{stmts_{n+1}},\sigma \\ \quad \rightsquigarrow_{\mathtt{P}} \mathtt{stmts_{n+1}},\sigma'\end{array}}$$

$$\frac{\mathtt{stmts},\sigma \rightsquigarrow_{\mathtt{P}} \mathtt{stmts}',\sigma'}{\begin{array}{l}\mathbf{try}\ \mathtt{stmts}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ \ldots\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n},\sigma \\ \quad \rightsquigarrow_{\mathtt{P}} \mathbf{try}\ \mathtt{stmts}'\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ \ldots\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n},\sigma' \\ \mathbf{try}\ \mathtt{stmts}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ \ldots\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ \mathbf{finally}\ \mathtt{stmts_{n+1}},\sigma \\ \quad \rightsquigarrow_{\mathtt{P}} \mathbf{try}\ \mathtt{stmts}'\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ \ldots\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ \mathbf{finally}\ \mathtt{stmts_{n+1}},\sigma'\end{array}}$$

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

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

Figure 30: Exception throwing, propagation and handling

3.8 Java$_r$ types

Java$_r$ terms corresponding to *normal execution* have a type, which is a combination of *normal type* and *abnormal type*. Java$_r$ terms corresponding to *abnormal execution* have a type, which is a *thrown type*.

The type of an address ($\iota_\mathtt{i}$) depends on the object or array pointed at in the current state σ; therefore, the *type* of a Java$_r$ term depends not only on the program and environment, but *also* the state, and this is why type assertions for Java$_r$ terms $\mathtt{t}$, executable on the state σ, have the followng forms:

- $\mathtt{D}, \Gamma, \sigma \vdash_{r/n} \mathtt{t} \ : \ \mathtt{T}$ – for the *normal type*;
- $\mathtt{D}, \Gamma, \sigma \vdash_{r/e} \mathtt{t} \ : \ \mathtt{ET}$ – for the *abnormal type*;
- $\mathtt{D}, \Gamma, \sigma \vdash_{r} \mathtt{t} \ : \ \mathtt{T} \parallel \mathtt{ET}$ – for the *type*, which is the combination of *normal type* and *abnormal type*;
- $\mathtt{D}, \Gamma, \sigma \vdash_{\overline{r}} \mathtt{t} \ : \ \mathtt{E\text{-}Thrn}$ – for the *type*, which is the *thrown type*.

Note the subscripts under the $\vdash$ which is used in order to distinguish various type systems and indicate the *normal type* and *abnormal type* inference in the Java$_r$ language.
The type rules for Java$_r$ *normal type*s are given in figure 31 and the type rules for Java$_r$ *abnormal type* are given in figure 32.

The Java$_r$ type rules correspond to Java$_{se}$ type rules, except where Java$_r$ introduces new syntax, or, where the subject reduction theorem proof requires otherwise. There are eight cases where Java$_r$ types differ from Java$_{se}$ types. The reasons for the differences can be classified into three categories.

Firstly, those that are related to an actually thrown exception. These are terms which contain the subterm **throw** $\iota_\mathtt{i}$, where $\iota_\mathtt{i}$ points to an object of an exception class. Such terms are only possible in Java$_r$ and not in Java$_{se}$. The type of these terms is *thrown type* of the form E-Thrn. The type rules are given in figure 34.

Secondly, those that give types to expressions that may only arise during program execution but do not involve exceptions. (*i.e.* the rules for addresses in figure 31). The terms in this category have the same type as if the address were replaced by an identifier of appropriate class or array type.

Thirdly, those that give types to terms that would be type incorrect in Java$_{se}$ (*i.e.* typing of assignments, the rules for null and for field access in figure 31). The rules in this category make type-correct terms which would have been type-incorrect in Java$_{se}$. However, the evaluation of such terms does not corrupt the integrity of the system, since unless infinite, it will throw an appropriate exception, if certain conditions are not satisfied.

Below we discuss these rules in more detail.

3.8.1 Normal types of Java$_r$

Figure 31 describes the *normal type*s for Java$_r$ terms.

There are two rules from the second category describing the types of addresses. Compared with figures 16 and 21, they can be found as additional rules on the fifth and sixth lines in figure 31.

If an object is stored at address $\iota_\mathtt{i}$, *i.e.* $\sigma(\iota_\mathtt{i}) = \ll ... \gg^\mathtt{C}$, then its class, C, is the type of $\iota_\mathtt{i}$. If a k-dimensional array of T is stored at such an address, *i.e.* $\sigma(\iota_\mathtt{i}) = [\![...]\!]^{\mathtt{T}[]_1...[]_k}$, then $\mathtt{T}[]_1...[]_k$ is the type of this reference.

The second rule on the first line, which belongs to the third category, says that null has any reference type. This rule is required in order to give a type to terms like $\mathtt{null[j-4]}$, which, although type incorrect in Java$_s$, may arise during execution of Java$_r$ terms. Such terms ultimately lead to exceptions, but they do not *immediately* raise the exception NullPE, because the Java semantics requires other parts of the expression to be evaluated first. In order to be able to prove the subject reduction theorem, such expressions need a type. The effect of this rule is, that Java$_r$ terms do not have unique types.

The rule for field access in Java$_r$ (the second one on the seventh line) is different from those in Java$_s$ and Java$_{se}$. Run-time value of field access may have the type which is different from that defined by the first visible field declaration as for Java$_s$ and Java$_{se}$. The type (*i.e.* $\mathtt{T}'$) for the

$$\frac{\mathtt{i}\text{ is integer},\quad \mathtt{c}\text{ is character},\quad \Gamma(\mathtt{x}) = \mathtt{T\ x}}{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{true} : \mathbf{bool},\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{false} : \mathbf{bool},\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{i} : \mathbf{int},\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{c} : \mathbf{char},\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{x} : \mathtt{T}\end{array}} \qquad \frac{\mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{Object}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{null} : \mathtt{T}}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e} : \mathbf{bool}\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{stmts} : \mathbf{void},\quad \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{stmts}' : \mathbf{void},\quad \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{stmt} : \mathtt{T}\end{array}}{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{stmts}\ ;\ \mathtt{stmt} : \mathtt{T}\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathbf{if}\ \mathtt{e}\ \mathbf{then}\ \mathtt{stmts}\ \mathbf{else}\ \mathtt{stmts}' : \mathbf{void}\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{v[e]} : \mathtt{T}\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e}' : \mathtt{T}'\\ \mathtt{D} \vdash \mathtt{T}' \leq_w \mathtt{T}\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{v[e]{=}e}' : \mathbf{void}} \qquad \frac{\begin{array}{l}\mathtt{v} \neq \mathtt{v}'[\mathtt{e}']\ \text{ for any } \mathtt{v}',\mathtt{e}'\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{v} : \mathtt{T},\quad \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e} : \mathtt{T}'\\ \mathit{NonFinalField}(\mathtt{D},\mathtt{v}),\ \mathtt{D} \vdash \mathtt{T}' \leq_w \mathtt{T}\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{v{=}e} : \mathbf{void}}$$

$$\frac{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e} : \mathtt{T}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathbf{return}\ \mathtt{e} : \mathtt{T}} \qquad \frac{}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathbf{return} : \mathbf{void}}$$

$$\frac{\begin{array}{l}\mathtt{D} \vdash \mathtt{C} \sqsubseteq \mathtt{C},\ \mathit{NonAbstractClass}(\mathtt{D},\mathtt{C})\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e_i} : \mathtt{T_i}\ \ \mathtt{i} \in \{1...\mathtt{n}\},\ \mathtt{n} \geq 1\\ \mathit{FirstFitConstr}(\mathtt{D},\mathtt{C},\mathtt{T_1} \times ... \times \mathtt{T_n}) = \{\mathtt{constrH}\}\\ \mathit{Mod}(\mathtt{constrH}) = \mathtt{cMod}\\ \mathtt{D},\Gamma \models_{s/n} \mathtt{this} : \mathtt{X},\ \mathit{Accessible}(\mathtt{D},\mathtt{X},\mathtt{C},\mathtt{cMod})\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathbf{new}\ [\mathtt{T_1} \times ... \times \mathtt{T_n}]\mathtt{C}(\mathtt{e_1}...\mathtt{e_n}) : \mathtt{C}} \qquad \frac{\sigma(\iota_\mathtt{i}) = \ll ... \gg^{\mathtt{C}}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \iota_\mathtt{i} : \mathtt{C}} \qquad \frac{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{v} : \mathtt{T[]}\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e} : \mathtt{int}\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{v[e]} : \mathtt{T}}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 1, \mathtt{k} \geq 0\\ \mathtt{D} \vdash \mathtt{T} \diamond_{VarType}\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e_i} : \mathtt{int}\ \ \mathtt{i} \in \{1...\mathtt{n}\}\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathbf{new}\ \mathtt{T}[\mathtt{e_1}]...[\mathtt{e_n}][]_1...[]_\mathtt{k} : \mathtt{T}[]_1...[]_{\mathtt{n+k}}} \qquad \frac{\sigma(\iota_\mathtt{i}) = [\![...]\!]^{\mathtt{T}[]_1 \cdots []_\mathtt{n}}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \iota_\mathtt{i} : \mathtt{T}[]_1...[]_\mathtt{n}}$$

$$\frac{\begin{array}{l}\vdash \mathtt{D} \diamond_a,\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{this} : \mathtt{C}\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{v} : \mathtt{T},\ \mathtt{D} \vdash \mathtt{T} \leq_w \mathtt{X}\\ \mathit{Accessible}(\mathtt{D},\mathtt{C},\mathtt{X},\mathtt{fMod})\\ (\mathtt{fMod\ C\ T'\ f}) \in \mathit{Fields}(\mathtt{D},\mathtt{T},\mathtt{f})\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{v.[X]f} : \mathtt{T}'} \qquad \frac{\begin{array}{l}\vdash \mathtt{D} \diamond_a,\quad \mathtt{D} \vdash \mathtt{X} \sqsubseteq \mathtt{X}\ \text{ or }\ \mathtt{D} \vdash \mathtt{X} \leq \mathtt{X}\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{this} : \mathtt{C},\ \mathbf{static} \in \mathtt{fMod}\\ \mathit{Accessible}(\mathtt{D},\mathtt{C},\mathtt{X},\mathtt{fMod})\\ \mathit{FSpec}(\mathtt{D},\mathtt{X},\mathtt{f}) = (\mathtt{fMod\ T\ f})\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{X.f} : \mathtt{T}}$$

$$\frac{\begin{array}{l}\vdash \mathtt{D} \diamond_a\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e_i} : \mathtt{T}'_\mathtt{i}\quad \mathtt{i} \in \{0...\mathtt{n}\}, \mathtt{n} \geq 0\\ \mathtt{D} \vdash \mathtt{T}'_\mathtt{i} \leq_w \mathtt{T_i}\quad \mathtt{i} \in \{1...\mathtt{n}\}\\ \mathit{FirstFit}(\mathtt{D},\mathtt{m},\mathtt{T}'_0,\mathtt{T_1} \times ... \times \mathtt{T_n}) = \{\mathtt{meth}\}\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e_0}.[\mathtt{T_1} \times ... \times \mathtt{T_n}\ ]\mathtt{m}(\mathtt{e_1}...\mathtt{e_n}) : \mathit{RetT}(\mathtt{meth})} \qquad \frac{\begin{array}{l}\mathtt{D} \vdash \mathtt{E} \sqsubseteq \mathtt{Exception}\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e} : \mathtt{E},\quad \mathtt{e} \neq \iota_\mathtt{i}\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/n} \mathbf{throw}\ \mathtt{e} : \mathbf{void}}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0\\ \mathtt{D} \vdash \mathtt{E_i}\ \sqsubseteq \mathtt{Exception}\quad \mathtt{i} \in \{1...\mathtt{n}\}\\ \mathtt{D} \vdash \Gamma \diamond\\ \Gamma(\mathtt{v_i}) = \mathit{Undef}\ \ \mathtt{i} \in \{1...\mathtt{n}\}\\ \mathtt{D},\Gamma;\ \mathtt{E_i\ v_i},\sigma \models_{r/n} \mathtt{stmts_i} : \mathbf{void}\ \ \mathtt{i} \in \{1...\mathtt{n}\}\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{stmts_0} : \mathbf{void}\ \ \mathit{OR}\ \ \mathtt{D},\Gamma,\sigma \models_r \mathtt{stmts_0} : \mathtt{E\text{-}Thrn}\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{stmts_{n+1}} : \mathbf{void}\end{array}}{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/n} \mathbf{try}\ \mathtt{stmts_0}\ \mathbf{catch}\ \mathtt{E_1\ v_1\ stmts_1}\ ...\ \mathbf{catch}\ \mathtt{E_n\ v_n\ stmts_n} : \mathbf{void}\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathbf{try}\ \mathtt{stmts_0}\ \mathbf{catch}\ \mathtt{E_1\ v_1\ stmts_1}\ ...\ \mathbf{catch}\ \mathtt{E_n\ v_n\ stmts_n}\\ \qquad \mathbf{finally}\ \mathtt{stmts_{n+1}} : \mathbf{void}\end{array}}$$

Figure 31: *Normal types* for Java$_r$ terms.

run-time value of field access depends on the actual type of the receiver (*i.e.* T) in addition to the declared type of the variable on the left of the field access (*i.e.* C) and may not be the first visible field already. The weaker requirement at this point is that run-time type of the receiver should have any declaration for the field of a type T′.

The rules on the third line, which belongs to the third category, describe assignments. The Java_r array assignment rule only requires the left hand side and the right hand side to be type-correct. It is weaker than the corresponding assignment type rule in Java_s or Java_{se}: it does not require the right hand side to be of a type that can be widened to that of the left hand side. The reason for this weaker requirement is, that the type of an array component may become narrower during evaluation.

3.8.2 Abnormal types of Java_r

Figure 32 describe the *abnormal type*s for Java_r terms. They are similar to the rules describing *abnormal type*s of Java_{se} terms in figures 17, 22 except for the rules concerned with cases for addresses (which are not part of Java_{se} syntax).

The *abnormal type* of a **throw** statement with address of an exception object is empty. So, the term **throw new** E, which indicates *potential* throwing of an exception, has the *abnormal type* E. This term is rewritten into **throw** $\iota_{\mathtt{i}}$, indicating an exception *actually* thrown, where $\iota_{\mathtt{i}}$ is the address of an object of class E, has empty *abnormal type*. The *abnormal type* of the exception context consists of the *abnormal type* of the term of the context.

3.8.3 Thrown types of Java_r

The rules in figure 34 deal with exceptions that have actually been thrown. Notice, that the term **throw new** E indicates *potential* throwing of an exception and has the *type* **void** $\|$ {E} according to the rules in figures 31 and 32. This term is rewritten to the term **throw** $\iota_{\mathtt{i}}$, where $\iota_{\mathtt{i}}$ is the address of an object of class E. The latter term indicates an exception which has *actually* been thrown, and, according to the rules, it has *no* the *normal type* and *abnormal type*, but has *thrown type* E-Thrn which is described by the first rule of figure 34. Particularly for such case in **try – catch** statement, the signal to switch to another type system is given in the alternative describing type of **try** statements in the last rule of both figures 31 and 32.

The *context* of an exception, defined in figure 33, encompasses all enclosing terms up to the nearest enclosing **try – catch** close, *i.e.* up to the first possible position at which the exception might be handled. According to the last rule in figure 34, the type of a term which is a context for a thrown exception of class E is E-Thrn. This rule allows the typing of a message expression one of whose arguments threw an exception, assignments whose left hand or right hand side threw an exception, *etc.*

The type E-Thrn describes a term in which an exception is thrown and not included inside a **try – catch** statement.

3.8.4 Properties of the Java_r type system

Definition 29 $\mathtt{D}, \Gamma, \sigma \models_r \mathtt{t} \ : \ \tau$ *iff*

- $\tau = \mathtt{T} \,\|\, \mathtt{ET}$ *and* $\mathtt{D}, \Gamma, \sigma \models_{r/n} \mathtt{t} \ : \ \mathtt{T}$ *and* $\mathtt{D}, \Gamma, \sigma \models_{r/e} \mathtt{t} \ : \ \mathtt{ET}$ *or*
- $\tau = \mathtt{E\text{-}Thrn}$ *and* $\mathtt{D}, \Gamma, \sigma \models_r \mathtt{t} \ : \ \mathtt{E\text{-}Thrn}$

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

The type E-Thrn characterizes Java_r terms that contain exceptions that has actually been thrown. Thus, the type E-Thrn can only be encountered when typing Java_r terms.

Definition 30 *For a* Java_r *term* t*, program* P*, declaration* D *with* $\mathtt{D} = \mathcal{D}(\mathtt{P})$*, environment* Γ *and state* σ *we define*

$$\frac{\mathtt{i}\text{ is integer},\quad \mathtt{c}\text{ is character},\quad \Gamma(\mathtt{x}) \neq \mathit{Undef}}{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/e} \mathbf{true} : \emptyset,\quad \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{false} : \emptyset\\ \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{i} : \emptyset,\quad \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{c} : \emptyset,\quad \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{x} : \emptyset\end{array}} \qquad \frac{}{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{null} : \emptyset}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{e} : \mathtt{ET}\\ \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{stmts} : \mathtt{ET}',\quad \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{stmts}' : \mathtt{ET}'',\quad \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{stmt} : \mathtt{ET}'''\end{array}}{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{stmts;stmt} : \mathtt{ET}'\cup\mathtt{ET}'''\\ \mathtt{D},\Gamma,\sigma \models_{r/e} \mathbf{if}\ \mathtt{e}\ \mathbf{then}\ \mathtt{stmts}\ \mathbf{else}\ \mathtt{stmts}' : \mathtt{ET}\cup\mathtt{ET}'\cup\mathtt{ET}''\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{v} : \mathtt{ET}\\ \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{e} : \mathtt{ET}'\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{v{=}e} : \mathtt{ET}\cup\mathtt{ET}'}$$

$$\frac{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{e} : \mathtt{ET}}{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathbf{return}\ \mathtt{e} : \mathtt{ET}} \qquad \frac{}{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathbf{return} : \emptyset} \qquad \frac{}{\mathtt{D},\Gamma,\sigma \models_{r/e} \iota_{\mathtt{i}} : \emptyset}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{e_i} : \mathtt{ET_i}\quad \mathtt{i}\in\{\mathtt{1...n}\},\ \mathtt{n}\geq 1,\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e_i} : \mathtt{T_i}\quad \mathtt{i}\in\{\mathtt{1...n}\},\ \mathtt{n}\geq 1,\\ \mathit{FirstFitConstr}(\mathtt{D},\mathtt{C},\mathtt{T_1}\times\ldots\times\mathtt{T_n}) = \{\mathtt{constrH}\}\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathbf{new}\ [\mathtt{T_1}\times\ldots\times\mathtt{T_n}]\mathtt{C}(\mathtt{e_1...e_n}) : (\cup\mathtt{ET_i})_{\mathtt{i=1}}^{\mathtt{n}}\cup \mathit{ExcT}(\mathtt{constrH})}$$

$$\frac{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{e_i} : \mathtt{ET_i},\qquad \mathtt{i}\in\{\mathtt{1...n}\}}{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathbf{new}\ \mathtt{T[e_1]...[e_n][]_1...[]_k} : (\cup\mathtt{ET_i})_{\mathtt{i=1}}^{\mathtt{n}}}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{v} : \mathtt{ET}\\ \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{e} : \mathtt{ET}'\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{v[e]} : \mathtt{ET}\cup\mathtt{ET}'} \qquad \frac{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{v} : \mathtt{ET}}{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{v.[C]f} : \mathtt{ET}} \qquad \frac{\begin{array}{c}\mathtt{D}\vdash\mathtt{X}\sqsubseteq\mathtt{X}\\ \text{or}\ \ \mathtt{D}\vdash\mathtt{X}\leq\mathtt{X}\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{X.f} : \emptyset}$$

$$\frac{\begin{array}{l}\vdash\ \mathtt{D}\diamond_a\\ \mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e_0} : \mathtt{T}_0'\\ \mathit{FirstFit}(\mathtt{D},\mathtt{m},\mathtt{T}_0',\mathtt{T_1}\times\ldots\times\mathtt{T_n}) = \{\mathtt{meth}\}\\ \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{e_i} : \mathtt{ET_i}\quad \mathtt{i}\in\{\mathtt{0...n}\}\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{e_0}.[\mathtt{T_1}\times\ldots\times\mathtt{T_n}]\mathtt{m}(\mathtt{e_1...e_n}) : (\cup\mathtt{ET_i})_{\mathtt{i=0}}^{\mathtt{n}}\cup \mathit{ExcT}(\mathtt{meth})}$$

$$\frac{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/n} \mathtt{e} : \mathtt{E},\qquad \mathtt{e}\neq\iota_{\mathtt{i}}\\ \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{e} : \mathtt{ET}\end{array}}{\mathtt{D},\Gamma,\sigma \models_{r/e} \mathbf{throw}\ \mathtt{e} : \{\mathtt{E}\}\cup\mathtt{ET}}$$

$$\frac{\begin{array}{l}\mathtt{n}\geq 0\\ \mathtt{D}\vdash\mathtt{E_i}\sqsubseteq\mathtt{Exception},\quad \mathtt{i}\in\{\mathtt{1...n}\}\\ \mathtt{D}\vdash\Gamma\diamond\\ \Gamma(\mathtt{v_i}) = \mathit{Undef}\quad \mathtt{i}\in\{\mathtt{1...n}\}\\ \mathtt{D},\Gamma;\ \mathtt{E_i}\ \mathtt{v_i},\sigma \models_{r/e} \mathtt{stmts_i} : \mathtt{ET_i},\quad \mathtt{i}\in\{\mathtt{1...n}\}\\ \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{stmts_0} : \mathtt{ET_0}\ \ \mathit{OR}\ \ (\mathtt{D},\Gamma,\sigma \models_{r} \mathtt{stmts_0} : \mathtt{E\text{-}Thrn}\ \ \mathit{AND}\ \ \mathtt{ET_0}=\emptyset)\\ \mathtt{D},\Gamma,\sigma \models_{r/e} \mathtt{stmts_{n+1}} : \mathtt{ET_{n+1}}\end{array}}{\begin{array}{l}\mathtt{D},\Gamma,\sigma \models_{r/e} \mathbf{try}\ \mathtt{stmts_0}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ldots\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n} :\\ \qquad\qquad\qquad (\mathtt{ET_0}\setminus_{\mathtt{D}}^{e}\{\mathtt{E_1},\ldots,\mathtt{E_n}\})\cup(\cup\mathtt{ET_i})_{\mathtt{i=1}}^{\mathtt{n}}\\ \mathtt{D},\Gamma,\sigma \models_{r/e} \mathbf{try}\ \mathtt{stmts_0}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ldots\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\\ \qquad \mathbf{finally}\ \mathtt{stmts_{n+1}} : (\mathtt{ET_0}\setminus_{\mathtt{D}}^{e}\{\mathtt{E_1},\ldots,\mathtt{E_n}\})\cup(\cup\mathtt{ET_i})_{\mathtt{i=1}}^{\mathtt{n+1}}\end{array}}$$

Figure 32: *Abnormal type*s for Java$_r$ terms.

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

Figure 33: Java_r exception contexts

$$\frac{\texttt{D} \vdash \texttt{E} \sqsubseteq \texttt{Exception} \qquad \sigma(\iota_{\texttt{i}}) = \ll \ldots \gg^{\texttt{E}}}{\texttt{D}, \Gamma, \sigma \models_r \textbf{throw}\ \iota_{\texttt{i}}\ :\ \texttt{E-Thrn}} \qquad \frac{\texttt{cont}\ @\cdot \mathrm{A}\ \text{ is a context} \qquad \texttt{D}, \Gamma, \sigma \models_r \texttt{t}\ :\ \texttt{E-Thrn}}{\texttt{D}, \Gamma, \sigma \models_r \texttt{cont}\ @\texttt{t}\,\mathrm{A}\ :\ \texttt{E-Thrn}}$$

Figure 34: *Thrown types* for Java_r terms.

$$\begin{array}{lcl}
\texttt{P}, \Gamma, \sigma \models_r \texttt{t}\ :\ \tau & \textit{iff} & \texttt{D}, \Gamma, \sigma \models_r \texttt{t}\ :\ \tau \\
\texttt{P}, \Gamma, \sigma \models_{r/n} \texttt{t}\ :\ \texttt{T} & \textit{iff} & \texttt{D}, \Gamma, \sigma \models_{r/n} \texttt{t}\ :\ \texttt{T}; \\
\texttt{P}, \Gamma, \sigma \models_{r/e} \texttt{t}\ :\ \texttt{ET} & \textit{iff} & \texttt{D}, \Gamma, \sigma \models_{r/e} \texttt{t}\ :\ \texttt{ET}
\end{array}$$

4 Extension: Class and Interface Modifiers

4.1 Description

Classes and interfaces are the largest sub-units of Java programs. They are permitted to have only one access modifier, viz. **public**. This is to allow access from different packages. The **abstract** modifier is another modifier that a class could have. An **abstract** class means that the class cannot be instantiated, maybe because the programmer wants to defer the implementation of some [**abstract**] methods. Since an interface cannot be instantiated, it is by definition already **abstract** so it is not necessary to explicitly mention the **abstract** modifier.

Also, if the programmer decides that a certain class' definition is complete and should not be modified by subclassing, it is possible to label the class as being **final**. This has the same effect as labeling all the methods as being **final** because there is no possibility of there being a subclass that can override and change the implementation of the methods.

5 Extension: Constructors

5.1 Description

Constructors are very similar to methods in the way they execute and in the way the most special constructor is resolved among overloaded constructors. In case of a class with multiple constructors, selection is done so on the basis of the arguments passed. These matching rules are exactly the same as the ones used during method invocations. Also, if no matching constructor is found, a compilation error is reported. At runtime, an invocation of a constructor results in the allocation of space for the variables of the given class with all the instance variables having initialisers being initialised, or it results in an OutOfMemory exception wherein execution halts even before the arguments for the constructor are evaluated. A constructor of a class looks like a method – the only difference is that constructors do not have a return type. Moreover, as constructors cannot be inherited like methods, there is no point in having either the **final** modifier or the **abstract** modifier. The main objective of having a constructor for a class is to permit the setup of the variables of the class when an object of that class type is instantiated. Only in case a constructor is not specified for a class, the compiler automatically inserts a default constructor with no arguments. If a certain class does not define a constructor with no arguments, but still has other explicitly defined constructors, it is very important for any subclasses to ensure that they do not make any explicit calls to **super()** and also that there is a **this** or **super(...)** explicit constructor invocation in every constructor of the subclases. Failure to comply with this will result in an attempt to invoke the non-existent no-argument constructor of the superclass, and a **java.lang.NoSuchMethodError** exception will result because the method: $< init > (Z)V$ cannot be found! Apparently, this is JDK's way of saying that there is no matching constructor. There is a condition enforced by the semantics which states that every class should define at least one constructor. In real Java, the compiler will insert the default constructor with no arguments, however, here, the programmer is expected to provide this default constructor if there are none others.

5.1.1 Execution Behaviour

- Space allocation
- Invocation of specified constructor uses same selection rules as for methods .. through unique constructor type (similar to method type)
- Explicit superclass constructor invocation **super(...)**
- Explicit same level constructor invocation **this(...)**
- Initialisation of instance variables (declared in current class only) with initialisers.
- Execution of statements in constructor body.

For a new object invocation expression, a constructor with a matching argument type has to be found, and it has to be verified that

- this constructor is accessible
- it belongs to a non-**abstract** class, not an interface

Figure 35 depicts the entry for the object of class `Dog` incarnated by the object invocation creation expression **new** `Dog();`. Reading the table from the top to bottom will show how the actual values for each field as they are stored in the entry change according to the execution of the program. Each row in the table shows the state at a certain point of execution of the program. When space is first allocated for the `Dog` object, the fields will not have any value assigned to them. However, before the first constructor starts executing, the fields are initialised with the default values which is determined by their types. The first statement in any constuctor is an explicit constructor invocation statement, and this in this case, it is an invocation of the constructor of `Dog` with an integer parameter. This one in turn invokes the `Dog` constructor with a boolean paramter. Finally, a constructor of the `Animal` class is invoked, and here too the explicit constructor invocation continues until the class `Object` is reached. After returning from constructor for `Object`, the initialisers for the fields of class `Animal` would be executed. None of the fields of class `Animal` has an initialiser, so all that happens at this level in the class hierarchy is that an assignment to a field are carried out by the `Animal` constructor with the **boolean**parameter. When the constructor call for the constructor of class `Animal` returns, the initialisers for the `siblings` and the `isHealthy` fields are executed and these two fields are

Representation of Dog object in state

Instance variables declared in class:	isHealthy Animal	:numLegs : Dog	:siblings : Dog	:isHealthy : Dog
Point in exectution		:	:	:
Allocation of space	—	: —	: —	: —
Initial values:	false	: 0	: false	: 0
Initialisation after super();		:	:	:
Animal();		:	:	:
Animal(**boolean**);	true	:	:	:
Initialisation after super();		: 4	:	: true
Dog(**boolean**);		:	:	: true
Dog(**int**);		:	: 0	:
Dog();		:	:	:
Finally	true	: 4	: 0	: true

Figure 35: Representation of state through execution

initialised. More assignments in the three constructors at the `Dog` level, and at the end, i.e. when the constructor finally returns, the entry for this object will be set up as is shown by the last row of Figure 35.

6 Extension: Field Modifiers

6.1 Description

A complete description of field modifiers, their types and their effects on the set of permissible actions.

6.1.1 Access Modifiers

Access modifiers of fields of a class determine the accessibility of those fields from outside the class. There are four different levels at which accessibility to the fields of a class can be controlled. The most important ones are complete accessibility (**public**) and complete inacessibility (**private**).

- ***public** * access modifier
  A field declared to be **public** in a class is accessible wherever the class itself is accessible. This means that the value represented by the field can be modified and/or read from any other part of the program. This feature is useful in cases where the data stored in the field is not of such paramount secrecy that it shouldn't be available outside the class. During execution, all attempts to access fields declared **public** are legal, and hence permitted by the compiler.

- ***protected** * access modifier

- ***package** * access modifier
  The absence of any means of representation of the notion of packages in the semantics in [1] and this paper as well means that there is no effective difference between the accessibility of **public** , **protected** and **package** access modifiers. Until semantics rules that can be used to reason about packages are devised there is no point in forming rules about **protected** and **package** access modifiers (See later).

  protected and default modifiers are similar to each other in the sense that the variables are visible only in the **package** they were declared in. However, **protected** variables are also visible in a class outside the **package** if that class is a subclass of the original class where the variables were declared **protected** .

- ***private** * access modifier
  A field can be completely hidden from any attempts to access it from outside the class by declaring it **private** . This means that the compiler will check for any illegal piece of code that attempts to either read the value stored in the variable, or even change it. Such a feature vastly diminishes the range of danger that can be caused by snippets of code from outside the class accessing and [secretly] modifying the data stored in the class. The **private** access modifier also helps Java to embody one of the most important features of object-oriented languages, viz. information-hiding, also known as encapsulation. This is of immense importance when it is necessary that the details of certain operations [as those carried out by the class] remain strictly **private** and are not disclosed under any circumstances. During execution, the compiler will catch any attemtpts to access **private** fields and generate a compilation error.

  A variable declared **private** will be visible only in the class in which it is declared.

6.1.2 Final Modifier

In the Java system, fields of a class declared **final** model the behaviour of constants more closely than that of variables, as should be the case. The way these work is that the fields declared with the **final** modifier will have an initialiser as part of their declaration. At run-time, the value that is assigned to the field when the object is instantiated will remain until the program completes execution, or the space allocated for that field is re-allocated. Any attempt to modify the value assigned to a final variable will result in a compilation error.

Constraints: have an initialiser, no assignment during execution, extension of program / environment to include information about initialisers.

6.1.3 Static Modifier

The fields of a class declared **static** are not really instance variables at all. Instead, such variables are shared among all instances of that object. These **static** variables, also known as class variables are initialised at the first active use [8] of the class. If the **static** variables have initialisers, these will be evaluated and consequently assigned to the variables only once during the whole period of execution of the program – this point too occurs at the first active use of the class. Since there is only one copy of a **static** variable, each update/assignment applied to the variable will be visible from any instance of the class. Moreover, the **static** variables of a class can be accessed without even having to create an instance of that object.

7 Extension: Method Modifiers

7.1 Description

A complete description of method modifiers, their types and their effects on the set of permissible method invocations.

7.1.1 Access Modifiers

Access modifiers of methods behave in exactly the same manner as those for variables – there are four different levels of accessibility, but as before, the lack of any means to represent packages makes it impossible to distinguish between the **public**, **protected** and **package** access modifiers.

***public * access modifier:** A method is accessible wherever the class itself is accessible.

***protected * access modifier** Effectively same as **public** .

***package * access modifier** Effectively same as **package** .

***private * access modifier** A method declared to be **private** is accessible only within the class it is declared in. This means that the method can be invoked by any other method, constructor and possibly **static** initialiser as long as it is from within the same class.

7.1.2 Final Modifier

A method declared with the final modifier means that it cannot be overridden in a subclass.

7.1.3 Abstract Modifier

Such a method will not have any implementation, and it also imposes an **abstract** modifier onto the class that the method is declared in.

8 Alternatives considered

8.1 Modifier information: storage

1. Power function: information retrieval
   Use a separate function with the sole intention of extracting information about the respective modifiers for fields, methods and constructors, given the declared class and type [argument type in the case of methods and constructors].

2. Extend **FDecs** and **MDecs**
   Modify the definitions of **FDecs** and **MDecs** to return the modifiers in addition to the type, declared class, etc. information that they were designed to return.

8.2 Identifying current method during execution

1. Single **this:C** in Γ

2. Nested **this:C** in Γ

8.3 Well-formedness with regards to access modifiers

1. Separate checking judgement
   Can only check for well-formedness after type-checking is done!

2. Extend typing judgement
   Check for well-formedness when type-checking.

8.4 Allocation of space for instantiation of objects

1. Power function: space allocation Allocation of space is done in a single function that makes use of **FDecs** to evaluate the total space required for the instantion of an object.

   - Pros
     Re-use of **FDecs**.
     Hidden implementation makes the whole allocation process transparent to the program.

2. Every class has a special anonymous method that is used to build up an intermediate data-structure as the call progresses up the **extends** hierarchy whereby the method calls its superclass' version of the special method as its first action.

3. Constructor: Automatic allocation The first step that any constructor does is to allocate space for all its locally declared variables.

   - Cons
     When more than one constructor is called at the same level, space is allocated multiple times.

9 Conclusions and Plans for the future.

I have managed to extend the semantics defined in [1] by a fair amount. The part of Java that still is not covered by the semantics is still considerably large, but I feel that with the ability to reason about the limitations that Java can exert on the programmer (*c.f.* accessibility and other issues relevant to **static** and **final** modifiers), what I have achieved has been worth it.

The research on Java semantics can be taken further still. There are still many aspects of Java yet uncovered by the semantics. Morever, Java is still changing, so until a point is reached when we can prove that a Java program will behave in exactly the same way while executing as it was specified to do so, research in this area will be ongoing.

10 Acknowledgements

I would like to express my heartfelt gratitude to my lecturers Sophia Drossopoulou, Susan Eisenbach and Krysia Broda at the Department of Computing, Imperial College for having introduced me to this field. Working with them in the summer of 1998 in the UROP project was what first got me interested in the semantics of Java. I am also very appreciative of the help and advice that they have offered me, both during the course of this research and before. Apart from that, I would like to thank the other members of the SLURP group, David, Mark, Tanya and Vishnu for their helpful feedback on many occasions.

References

[1] S. Drossopoulou, S. Eisenbach and T. Valkevych. Java Types Soundness Extended to Exceptions, June 1999.

[2] S. Drossopoulou and S. Eisenbach. Towards an Operational Semantics and Proof of Type Soundness for Java . In *Proceedings of the European Conference on Object-Oriented Programming*, June 1997.

[3] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java Type System Sound? *Theory And Practice of Object Systems*, 5(1):3–24, 1999.

[4] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A Fragment Calculus – Towards a Model of Separate Compilation, Linking and Binary Compatibility. In *LICS'99 Proceedings*, 1999. to appear.

[5] Sophia Drossopoulou and Susan Eisenbach. Java is Type Safe — Probably. In *Proceedings of the European Conference on Object-Oriented Programming*, June 1997.

[6] Sarfraz Khurshid. Some Aspects of Type Soundness for Java, 1997. BSc thesis.

[7] The Java Language Specification, May 1996.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, August 1996.

[9] Leslie Lamport. *LaTeX – A Document Preparation System.* Addison-Wesley, 1985.