A Core Calculus for Java Exceptions*

(Extended Abstract)

Davide Ancona, Giovanni Lagorio, and Elena Zucca

DISI - Università di Genova
Via Dodecaneso, 35, 16146 Genova (Italy)
email: {davide,lagorio,zucca}@disi.unige.it

Abstract. In this paper we present a simple calculus (called CJE) corresponding to a small functional fragment of Java supporting exceptions. We provide a reduction semantics for the calculus together with two equivalent type systems; the former corresponds to the specification given in [5] and its formalization in [4], whereas the latter can be considered an optimization of the former where only the minimal type information about classes/interfaces and methods are collected in order to type-check a program. The two type systems are proved to be equivalent and a subject reduction theorem is given.

1 Introduction

The aim of this paper is to deeply investigate the exception mechanism of Java (in particular its interaction with inheritance) by means of a simple calculus, called CJE for Calculus of Java Exceptions.

Although calculi for Java exceptions have been considered already in [3, 2], much still has to be said about this topic; indeed, the approach taken here originates from [2] and presents several interesting novelties in comparison with [3].

First, we have deliberately omitted from the calculus all the features which we consider orthogonal w.r.t. the exception mechanism in Java. Indeed here, following the approach of Featherweight Java (abbreviated FJ) in [6], the idea is to keep the calculus as simple as possible rather than trying to give a soundness result for the whole Java language.

Even though it is certainly useful to have a full formal description of the semantics of Java (and [4, 3] go toward this direction), we think that, when proving specific properties or trying to clarify some tricky point in the informal specification, it is better to separate the concerns and to consider an essential subset of the language. As a result, the subject reduction proof we give for CJE is rather simple and compact.

Since in this paper we want to focus on Java exception handling, CJE clearly includes all the linguistic mechanisms for handling exceptions (`finally` excluded[1]), but many others are omitted; among them, fields, method overloading (even though not completely, see the discussion below about abstract classes), constructors, `super` and even assignment; indeed, like FJ, CJE has no statements (hence is a functional language). More precisely, the calculus has only four constructs: method invocation, object creation and the `throw`, `try` and `catch` expressions.

On the other hand, we have not omitted abstract classes since there are some subtle points in the rules of the Java language specification [5] for checking conflicts in the `throws` clause and such rules are particularly complex for abstract classes. These problems do not emerge from the formalization given in [4, 3] where abstract classes are not considered.

Second, we are interested in another kind of simplification which corresponds to minimizing the information associated with classes and interfaces needed in order to type-check a program. Indeed, the type information about classes and interfaces used in the type system defined in [4, 3]

* Partially supported by Murst - TOSCA Teoria della Concorrenza, Linguaggi di Ordine Superiore e Strutture di Tipi and APPlied SEMantics - Esprit Working Group 26142.

[1] The `finally` clause makes no sense in a functional setting; on the other hand, the extension of the type system to include such mechanism does not imply any problem.

is somehow redundant. Avoiding such redundancy makes, in our opinion, the type system clearer and helps to shed some light on the obscure points of the Java specification given in [5] (which can be easily misinterpreted, as shown in the example in Sect.2).

However, this kind of minimization, even though advocated for sake of clarity, neither necessarily implies a simplification of the proofs, nor aims at making type-checking of Java more efficient (even though, in principle, it could be the case and this matter deserves further investigation).

In Sect.2 we give a paradigmatic and motivating example used for showing the complexity of the rules for the compatibility checks of `throws` clauses and for explaining the notion of minimal type. Sect.3 is an outline of the formal definition of the calculus and of the main technical results. Finally, in Sect.4 we draw some conclusion and claim that some modification to the Java language could partly avoid the complexity of the rules.

An extended version of this paper can be found in [1].

2 A paradigmatic example in Java

Consider the following Java code fragment:

```
interface I {
 void m1() throws E1;
 void m2() throws E1;
 void m3();
}
abstract class C1 {
 public abstract void m1() throws E0,E2;
 public void m2() throws E0{}
 public void m3() throws E0{}
}
abstract class C2 extends C1 implements I {
 public abstract void m3();
}
```

If we assume that `E0`, `E1` and `E2` are exception types s.t. `E0` $\leq_c^\rho$ `E1` $\leq_c^\rho$ `E2` (where $\leq_c^\rho$ corresponds to the subclass relation determined by an environment ρ of classes/interfaces corresponding to a Java program; see Sect.3), then the declaration of the class `C2` is statically correct. Indeed, according to [5] 8.4.6.4, methods `m1` of `I` and `m1` of `C1` are both inherited by `C2` and no compatibility check for the `throws` clauses is required; note that if `m1` were not abstract in `C1` then the code would be not correct. Indeed, in that case `m1` in `C1` would override, and therefore implement, `m1` in `I` and a compatibility check for the `throws` clauses (which clearly would fail since `E2`$\not\leq_c^\rho$`E1`) would be required.

On the other hand, `m2` in `C1` overrides `m2` in `I`, therefore the compatibility check has to be performed and in this case it is passed since `E0` $\leq_c^\rho$ `E1`.

Finally, `m3` in `C2` overrides `m3` in `I` and `C1`, hence the two corresponding checks have to be performed, whereas no check is needed between `m3` in `I` and `m3` in `C1`; note that this last check would be performed and would fail if `m3` were not in `C2`.

From this example it should be clear that a type system modeling compatibility checks for exceptions in presence of an inheritance hierarchy including interfaces and abstract classes is really needed. Indeed, such a type system would provide a formal basis for understanding what is going on and whether these complex rules are really necessary or, rather, it is better to take a simpler approach by somehow restricting the language, as briefly discussed in Sect.4.

Let us now show what we mean by simplification of types. According to [5] and its formalization in [4, 3], the type information associated with the class `C2` can be expressed as follows:

```
{void m1() throws E1; void m1() throws E0,E2;
 void m2() throws E0; void m3()}
```

First, we can notice that the clause `throws E0,E2` contains some redundancy, since it is equivalent to `throws E2`, by virtue of the hypothesis `E0` $\leq_c^\rho$ `E2`; hence, we can apply a simplification step to the type above obtaining a new type where all the `throws` clauses are minimal.

The other redundancy is that the method `void m1()` is repeated twice, whereas for type-checking classes/interfaces it is enough to have a unique occurence where the `throws` clause is obtained by means of a sort of "intersection" operator (which formally corresponds to take the greatest lower bound w.r.t. the natural order for exception sets); therefore, applying this second simplification step we obtain the minimal type

```
{void m1() throws E1; void m2() throws E0; void m3()}.
```

3 Formal definitions and results

The abstract syntax and the reduction semantics of CJE are given in Fig.1 and 2, respectively.

In the syntax we use the notation A^* to indicate a sequence of zero or more occurrences of A and $A^{\circledast}$ (resp. $A^{\oplus}$) to indicate a set (resp. non empty set) of occurrences of A, that is, a sequence in which there are no repetitions and the order is immaterial. The terminals `iname` and `cname` indicate interface and class names respectively. A generic name is indicated by `name`.

Note that since CJE is a functional language, the `throw` and the `try` and `catch` constructs are not statements, as happens in Java, but expressions; furthermore, for sake of simplicity, the `throw` expression is built on top of class names (corresponding to exception names) rather than generic expressions.

The metavariable E ranges over *expr*, C over class names and m over method names.

There are two kinds of normal forms: those having form `new` C, corresponding to normal program terminations evaluating into an object of class C, and those of the form `throw` C, corresponding to abnormal program terminations throwing the exception C.

The reduction relation $\rightarrow_\rho$, the auxiliary function $Body_\rho$ and the subclass relation $\leq_c^\rho$ are all indexed by ρ which is the environment of classes/interfaces w.r.t. which the reduction is performed; more precisely, ρ represents a Java program P and associates with any class/interface name the corresponding declaration in P.

The auxiliary function $Body_\rho$, whose definition has been omitted for lack of space, performs method look-up in the environment ρ: $Body_\rho(C, m)$ returns the tuple $\langle E, x_1 \ldots x_n \rangle$ corresponding to the body and the formal parameters, respectively, of the method named m found when starting look-up at the class C in the environment ρ; if the method is not found, then $Body_\rho(C, m)$ is undefined.

Two different type systems are considered. The former, which we call *full*, corresponds to the specification given in [5] and formalized in [4, 3], whereas the latter, which we call *minimal*, uses minimal types. For lack of space Fig.3 contains only the rules for type assignment to classes/interfaces (see [1] for the complete set of rules).

The type of a class/interface is a pair consisting of a *fields type* and a *methods type*. A fields type is a set of fields, that is, pairs consisting of a field name and a field type; a methods type is a set of methods, that is, pairs consisting of a method name qualified by the types of the arguments (what is usually called a *signature*) and a triple consisting of the *kind* (abstract or not abstract), the return type and the set of declared exceptions.

The rules in Fig.3 have the same structure for both the full and the minimal type systems. What changes is the definition of the auxiliary *sum* functions $\overset{\Gamma}{\oplus}$ used for defining the rules (see below).

The first rule defines the type of an interface I, which consists of an empty fields type and a methods type which is the sum of the methods types of the direct superinterfaces (I' s.t. $I <_i^1 I'$ in Γ), updated by the methods MST declared in I.

The second rule defines the type of a class C. The field types consists in the fields type FST' of the direct superclass updated by the fields FST declared in C. The methods type consists of the sum of the methods types of the implemented interfaces (I s.t. $C \lhd_i^1 I$ in Γ) and the abstract

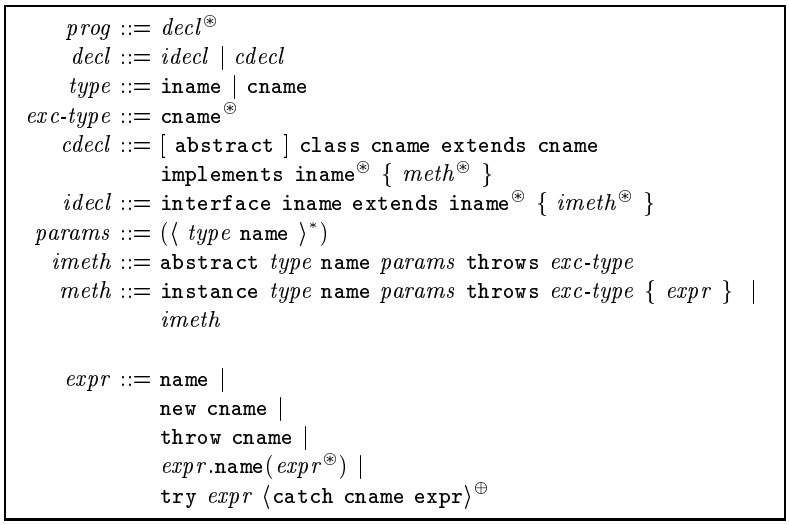

Fig. 1. Syntax

methods of the superclass C', updated by the non abstract methods of C' updated in turn by the methods MST declared in C. The last side condition expresses the constraint that a class with abstract methods must be declared abstract (see [5] 8.4.3.1).

The auxiliary *update* operations on fields and methods types, written $\_[\_]$, return a new fields (resp. methods) type obtained updating the first argument with new fields (resp. methods), if this is possible, accordingly with Java rules on hiding, overloading and overriding (see [5] 8.4.6 and 8.4.7). For instance, updating a methods type is undefined if we try to override a method with another which has the same signature and incompatible `throws` clause.

The sum operation $\overset{\Gamma}{\oplus}$ is just set union in the full type system, whereas in the minimal type system it is a modified union that can merge method types as illustrated in Sect.2. More precisely, merging two method types having ES and ES' as sets of declared exceptions, respectively, produces just one method type whose set of declared exceptions is the "intersection" of ES and ES' defined by

$C \in ES \overset{\Gamma}{\otimes} ES'$ iff either $C \in ES$ and $\exists C' \in ES'$ *s.t.* $\Gamma \vdash C \leq C'$ or conversely.

When trying to sum two methods with the same signature, the operation is defined (in both versions) only if such methods have the same return type.

The two auxiliary functions *Abstract* and *NonAbstract* return the set containing only the abstract and non abstract methods, respectively.

For the formal definition of update and sum operations see [1].

In order to state that the full and minimal type systems are equivalent, for each type environment Γ (containing all the type information about the classes and interfaces in a program) we define a function $simp_\Gamma$: *FullTypes* $\rightarrow$ *MinTypes* which, given a full type τ, returns its simplified version $simp_\Gamma(\tau)$, that is, a minimal type. This function is indexed by Γ since the simplification depends on the subclass and subinterface relations which holds for a given program, as explained in Sect.2.

Actually, the function $simp_\Gamma$ is a logical relation $\sim_\Gamma \subseteq$ *FullTypes* $\times$ *MinTypes* defined by $\tau \sim_\Gamma \tau'$ iff $simp_\Gamma(\tau) = \tau'$. This relation is used in the following theorem stating the equivalence of the full and minimal type systems.

Theorem 1 (Equivalence of the Type Systems). *For any type environment Γ, CJE expression E and type τ, if $\Gamma \vdash_f E{:}\tau$ then there exists τ' s.t. $\Gamma \vdash_m E{:}\tau'$ and $\tau \sim_\Gamma \tau'$. Conversely, if $\Gamma \vdash_m E{:}\tau$ then there exists τ' s.t. $\Gamma \vdash_f E{:}\tau'$ and $\tau' \sim_\Gamma \tau$.*

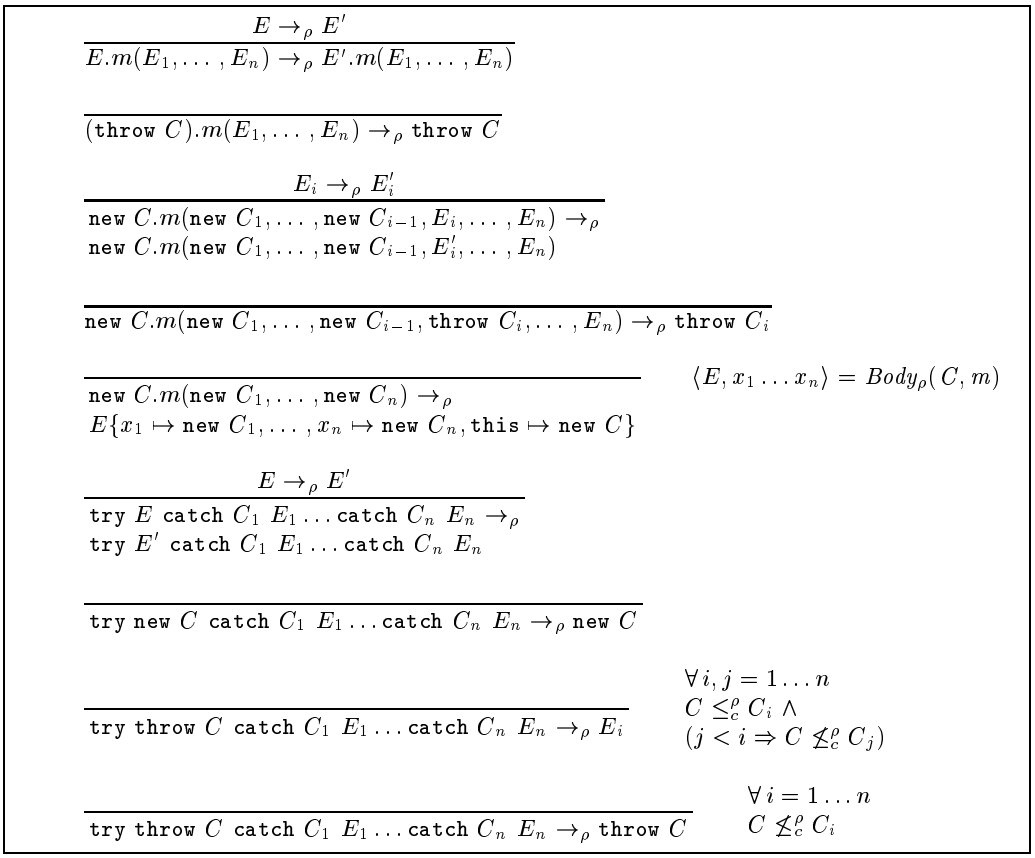

Fig. 2. Reduction rules

Here $\Gamma \vdash_f E{:}\tau$ and $\Gamma \vdash_m E{:}\tau'$ are the usual typing judgments for expressions in the full and the minimal type system, respectively.

As usual, the theorem can be proved by induction on the typing rules and by proving analogous properties for all the other judgments of the system. For instance we expect also the following property to hold: for any type environment Γ, $\Gamma \vdash_f \diamond$ iff $\Gamma \vdash_m \diamond$, where $\Gamma \vdash_f \diamond$ and $\Gamma \vdash_m \diamond$ are the judgments for well-formed type environments in the full and the minimal type system, respectively.

Another important property that we prove is subject reduction.

Theorem 2 (Subject Reduction). *For any type environment Γ, class/interface environment ρ, CJE expressions E, E' and type τ, if $\Gamma \vdash_f E{:}\tau$, $\Gamma \vdash_f \rho\ \diamond$ and $E \to_\rho E'$, then there exists a type τ' s.t. $\Gamma \vdash_f E'{:}\tau'$ and $\Gamma \vdash_f \tau' \leq \tau$.*

Here $\Gamma \vdash_f \rho\ \diamond$ is the judgment for well-formed class/interface environments ρ w.r.t. a given type environment Γ.

The subject reduction property for the simplified system follows by virtue of the equivalence stated in Theorem 1 and by the following two properties:

- $\Gamma \vdash_m \rho\ \diamond$ implies $\Gamma \vdash_f \rho\ \diamond$;
- $\Gamma \vdash_f \tau_f' \leq \tau_f$, $\tau_f \sim_\Gamma \tau_m$, $\tau_f' \sim_\Gamma \tau_m'$ implies $\Gamma \vdash_m \tau_m' \leq \tau_m$.

4 Conclusion

We have presented a core calculus for Java exceptions, defined its reduction semantics and two provably equivalent type systems, and proved subject reduction for it.

$$\frac{\Gamma \vdash I \text{ is}_i\ MST \quad \Gamma \vdash MST \diamond_{\text{InterfaceType}} \qquad \Gamma \vdash I_1 : \emptyset\ MST_1 \ldots \Gamma \vdash I_n : \emptyset\ MST_n}{\Gamma \vdash I : \emptyset\ (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n)[MST]_\Gamma} \quad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \{I' | I <_i^1 I' \in \Gamma\} \end{array}$$

$$\text{Set } MST_c = (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n \overset{\Gamma}{\oplus} Abstract(MST'))[NonAbstract(MST')[MST]_\Gamma]_\Gamma,$$

$$\frac{\begin{array}{l} \Gamma \vdash C \text{ is}_c\ K\ FST\ MST \\ \Gamma \vdash CT \diamond_{\text{ClassType}} \\ \Gamma \vdash C <_c^1 C' \\ \Gamma \vdash C' : FST'\ MST' \\ \Gamma \vdash I_1 : \emptyset\ MST_1 \ldots \Gamma \vdash I_n : \emptyset\ MST_n \end{array}}{\Gamma \vdash C : FST'[FST]\ MST_c} \quad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \{I | C \lhd_i^1 I \in \Gamma\} \\ K = \texttt{concrete} \Rightarrow Kind(MST_c) = \texttt{concrete} \end{array}$$

Fig. 3. Type assignments to classes/interfaces for both systems

The first important contribution of this paper is the full formalization of the complex rules given in [5] for compatibility checks of `throws` clauses; such rules have to be performed whenever a class/interface is extended (by inheritance) and are particularly nasty when abstract classes and implementation of interfaces are involved.

The second contribution is the definition of a minimal type system which is proved to be equivalent to that given in [4] and has the advantage of avoiding redundancies in favor of a better understanding of exception handling in Java.

Furthermore, our analysis has pointed out that in some cases the rules defining the static correctness of Java programs can be very tricky. Our feeling is that this could be avoided at the cost of a minimal loss of flexibility, by adding some restriction to the language, as sketched below.

In Java if a class C is declared to implement an interface I, then all methods in I that are neither defined in C nor inherited from the superclasses of C are implicitly inherited by C. This rule implies that a class can inherit more methods with the same signature, a rather strange situation in a language where classes cannot have more than one parent.

A more coherent choice would consist in requiring C to have all methods (either defined or inherited from its superclasses) contained in I, with the consequence that C cannot inherit methods from I but only implement them (as properly suggested by the keyword `implement`). Following this approach, we would also avoid the counter-intuitive Java rule stating that an abstract method m in C implements a method m in I only if m is directly defined in C (see [5] 8.4.6.1 and 8.4.6.4).

References

1. D. Ancona, G. Lagorio, and E. Zucca. A core calculus for Java exceptions. Technical report, DISI, University of Genova, 2000. In preparation.
2. D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of Java with mixins. In E. Bertino, editor, *ECOOP 2000 - European Conference for Object-Oriented Programming*, Lecture Notes in Computer Science. Springer Verlag, 2000. To appear.
3. S. Drossopoulou and T. Valkevych. Java exceptions throw no surprises. Technical report, Dept. of Computing - Imperial College of Science, Technology and Medicine, March 2000.
4. S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Technical report, Dept. of Computing - Imperial College of Science, Technology and Medicine, October 1999.
5. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
6. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.