A Fully Abstract Trace-based Semantics for Reasoning About Backward Compatibility of Class Libraries[☆]

Yannick Welsch[*], Arnd Poetzsch-Heffter[*]

Department of Computer Science, University of Kaiserslautern, Germany

Abstract

Proving that a class library is backward compatible to an older version can be challenging, as the internal representation of the libraries might completely differ and the clients of the library are usually unknown. This is especially difficult in the setting of object-oriented programs with complex heaps and callbacks.

In this paper, we develop a fully abstract trace-based semantics for class libraries in object-oriented languages, in particular for Java-like sealed packages. Our approach enhances a standard operational semantics such that the change of control between the library and the client context is made explicit in terms of interaction labels. By using traces over these labels, we abstract from the data representation in the heap, support class hiding, and provide fully abstract package denotations. Soundness and completeness of the trace semantics is proven using specialized simulation relations on the enhanced operational semantics. The simulation relations also provide a proof method for reasoning about backward compatibility.

Keywords: full abstraction, class libraries, trace semantics, contextual equivalence, backward compatibility

1. Introduction

Object-oriented libraries are usually realized by the complex interplay of different classes. As libraries evolve over time, adaptations have to be made to their implementations. Sometimes such evolution steps do not preserve backward compatibility with existing clients (called *breaking API changes* in [1]), but often libraries should be modified, extended, or refactored in such a way that client code is not affected. Software developers can use informal guidelines (e.g., [2]) and special tools (e.g., [3]) to check compatibility aspects. Reasoning about the behavioral equivalence of two library implementations is not only useful when no specifications are available; we conjecture (similar to Godlin and Strichman [4]) that it may often be simpler to verify the behavioral equivalence of two similar library implementations than to build a specification of the full behavior of the library and verify conformance to the specification.

Mutual compatibility corresponds to the classical notion of *(contextual) equivalence*: Two class library implementations are equivalent if they exhibit the same operational behavior in every possible class context. Proving backward compatibility or equivalence is challenging because (1) the possible contexts are unknown and complex, and (2) the stacks and heaps can be significantly different between the library versions. To meet these challenges, we exploit denotational methods. A denotational semantics for classes is called *fully abstract* [5, 6] if classes that have the same denotation are exactly those that are contextually equivalent. In particular, a fully abstract semantics has to abstract from stacks and heaps to meet challenge (2) above. Proving that two sets of classes are equivalent in the (fully abstract) denotational setting amounts to proving that they have the same denotation.

The central contributions of this paper are the design of such a fully abstract semantics for packages of a Java subset, a detailed explanation of the full abstraction proof and a method for reasoning about backward compatibility using specialized simulation relations.

[☆]This work has been partially supported by the EU project FP7-231620 HATS.
[*]Corresponding author. Tel: +49 631 205 2623. Fax: +49 631 205 3420.
Email addresses: `welsch@cs.uni-kl.de` (Yannick Welsch), `poetzsch@cs.uni-kl.de` (Arnd Poetzsch-Heffter)

```
public class Cell { // Version 1
  private Object c;
  public void set(Object o) { c = o; }
  public Object get() { return c; }
}
```

```
public class Cell { // Version 2
  private Object c1, c2; private boolean f;
  public void set(Object o) {
    f = !f; if (f) c1 = o; else c2 = o; }
  public Object get() { return f ? c1 : c2; }
}
```

Figure 1: Cell example

To illustrate the issues addressed by this paper, let us consider a very simple library at the left of Figure 1 which provides a Cell class to store and retrieve references to objects. In a more refined version of the Cell library on the right of Figure 1, a library developer might now want the possibility to not only retrieve the last value that was stored, but also the previous value. In the new implementation of the class, the developer therefore introduces two fields to store values and a boolean flag to determine which of the two fields stores the last value that was set. This second representation allows to add a method to retrieve not only the last value that was stored, but also the previous one, e.g. `public Object getPrevious(){ return f ? c2 : c1; }`.

The developer might now wonder whether the old version of the library can be safely replaced with the new version, i.e., whether the new version of the Cell library still retains the behavior of the old version when used in program contexts of the old version. Intuitively, the developer might argue in the following way why he believes that the new library version is backward compatible to the old one: If the boolean flag in the new library version is true, then the value that is stored in the field c1 corresponds to the value that is stored in the field c in the old library version. Similarly, if the boolean flag is false, then the value that is stored in c2 corresponds to the value that is stored in c. In the remainder of the paper, we give a formal underpinning to this intuition and present a solution to formally reason about such properties.

1.1. Approach

In our approach, the denotation of a library (set of classes and interfaces) is expressed by the interactions between code belonging to the library and code belonging to the program context. It is defined in *two* steps starting from a standard operational semantics. In the first step, the operational semantics is augmented in a way that the interactions can be made explicit. In the second step, traces of interaction labels are used to semantically characterize the library behavior. A non-trivial aspect is the treatment of inheritance, because with inheritance, some code parts of a class/object might belong to the context and other parts to the library under investigation.

Using traces allows abstracting from the state and heap representation in the old and new version. It solves challenge (2). To obtain a finite representation of all contexts and solve challenge (1), we construct a nondeterministic *most general context* that exactly exhibits the possible behavior of contexts. Using an operational semantics as a starting point has the advantage that we can use simulation relations applied to standard configurations (i.e., heap, stack) for the full abstraction proof and as a reasoning method for backward compatibility. Furthermore, it provides a direct formal relation to Hoare-like program logics and standard techniques for static program analysis.

To illustrate the intricacies of the trace semantics that goes beyond the simple Cell example in Figure 1, we consider a simple utility library in Figure 2 that provides classes to implement the Subject/Observer pattern. The observers, which are implementing the Observer interface, can be added to the Subject using the addObserver method and are stored in a linked list. The Subject also offers the possibility to get an iterator, basically a cursor, to navigate over the list of registered observers. Furthermore, the Subject provides the convenience method notifyObservers to update all the registered observers with the given argument. The example illustrates some of the difficulties when dealing with representation independence of OO libraries:

Multiple roles: The LinkedList class can be used directly by the context (as it is public) as well as serve as an internal representation of the Subject class.

Complex interface: There may be multiple objects (e.g., iterators) available to the context that access shared internal data of the library (e.g., the LinkedList representation).

```
1   public interface Observer {
2     public void update(Object arg);
3   }
4   public class Subject {
5     private LinkedList obs;
6     public void addObserver(Observer o) { ... }
7     public Iterator iterator() {
8       return new ObsIter();
9     }
10    public void notifyObservers(Object arg) {
11      while ( ... ) { ... o.update(arg); ... }
12    }
13    ...
14    private class ObsIter implements Iterator {
15      private int currIdx;
16      ObsIter() { ... }
17      public boolean hasNext() { ... }
18      public Object next() { ... }
19    }
20  }
21  public interface Iterator {
22    public boolean hasNext();
23    public Object next();
24  }
25  public class LinkedList { ... }
```

Figure 2: Observer example

```
1   public class IntObs implements Observer {
2     private int count = 0;
3     public void update(Object arg) {
4       count += ((Integer)arg).intValue();
5   } }
6   // Body of main method
7   Subject sj = new Subject();
8   Observer ob = new IntObs();
9   sj.addObserver(ob);
10  sj.notifyObservers(new Integer(5));
```

Figure 3: Program context for Observer example

Type abstraction: The Iterator interface allows abstracting from internal class implementations, i.e., clients of the library do not need to be aware of the implementation type of the iterator. Vice-versa, the library does not need to know about possible classes in the program context implementing the Observer interface.

Callbacks: During a notification, i.e., call of the method update, an observer can make a callback to the Subject under consideration.

In order to abstract from the complex representation of the library (heap and stack configurations), we describe the library in terms of its "input/output" behavior. The main question to address is what we consider as the points where such observable behavior occurs and what the input/output information is. As we are in a sequential setting, control flow can at a fixed point in the execution either be in code of the library or code of the program context. The points of observation thus become those where control flow changes from the library to the context and vice-versa. The behavior of a program context with a library is described by a trace, i.e., a sequence of labels that record the input/output between the context and the library. The form of the input/output labels is chosen such that they capture all relevant information about the behavior of the library.

To illustrate how these traces look like, let us consider the program context in Figure 3, which consists of an implementation of the Observer interface and a main method which uses the Subject. The traces which are generated by the program that consists of this program context and the utility library in Figure 2 are of the form

call o_1.addObserver(o_2)⍗ · rtrn _⍐ · call o_1.notifyObservers(o_3)⍗ · call o_2.update(o_3)⍐ · rtrn _⍗ · rtrn _⍐

where o_1, o_2, o_3 are arbitrary but distinct object identifiers[1]. In the following, we describe how this trace is constructed.

Program execution starts at line 6 (beginning of the main method) of the program context in Figure 3. Assuming default constructors, the first change in control happens at line 9, where the method addObserver is called. Execution jumps to the beginning of the body of this method, which is at line 6 in Figure 2. Due to the

[1]To simplify the presentation for this example, we have omitted some of the information in the trace regarding types.

change in control from the program context to the library, the input label call o_1.addObserver(o_2)⍗ is recorded. It contains the information that we have a method call of the method addObserver, that two distinct objects o_1 and o_2 are callee and parameter of the method call and that the direction of the change in control is from the program context to the library, which is denoted by ⍗ for input. Within the body of the addObserver method (which is not shown), the observer is added to the list of observers and the method returns. When the method returns, we have again a change in control, but this time in the reverse direction. The output label rtrn _⍐ is thus recorded, which contains the information that the change in control is due to a method return, that no values are passed (void method) and ⍐ denotes that this is an output label. We are now back executing in the program context and execute the next statement in line 10 of Figure 3 by calling the method notifyObservers. This leads to the label call o_1.notifyObservers(o_3)⍗, which contains the information that the method is called on the same object o_1 that we called addObserver before. In the body of the notifyObservers method at line 11 in Figure 2, the program iterates over the (singleton) list of registered observers and calls the update method. This leads again to a change in control as the update method for this particular observer has been defined in the program context. This is recorded by the label call o_2.update(o_3)⍐ that shows exactly which objects are involved in this call. Finally the update method returns (rtrn _⍗), then the notifyObservers method returns (rtrn _⍐) and the main method terminates.

The characterization of the behavior of a library (with a program context) in terms of traces is independent of the operational representation of the library (i.e. heaps and stacks). However, we want to describe the behavior of a library not only in terms of a specific program context, but of all possible program contexts.

We introduce a *most general context* (MGC) that enables all interactions that a standard program context can engage in. Compared to a standard program context, the most general context abstracts over types, objects and operational steps. We recapitulate the capabilities of standard program contexts before we describe the abstraction realized by the MGC. In terms of program code, program contexts can define classes that rely on the library, either by calling code of the library or by extending classes and implementing interfaces of the library (e.g. class IntObs in Figure 3). In terms of operational steps, program contexts can (1) create objects of classes that are defined in the program context (e.g. IntObs in line 8 of Figure 3) or objects of classes that are defined in the library and accessible to the program context (e.g. Subject in line 7, but not ObsIter). (2) perform steps that do not lead to a change in control, e.g., access and write fields and method calls/returns that are dispatched to code of the program context (e.g. line 4 in Figure 3). The MGC abstracts over these steps. (3) call methods that are defined in the library (e.g. line 9 in Figure 3) or return to code that is defined in the library (e.g. line 5 in Figure 3).

The only relevant actions which should be done by the MGC are those that lead to a change in control. In short, the MGC can call methods using available objects or simply return to method invocations from the library. Available objects are either those that are created by the MGC or the ones that have been leaked by the library.

1.2. Related Work

There is a large body of work on studying the equivalence of programs and program parts. In this presentation, we focus on such studies in the setting of object-oriented programs. The most popular way to reason about the behavior of class implementations is to use denotational methods, bisimulations or both.

Denotational methods have been successfully used to investigate properties of object-oriented programs [7]. The denotational semantics according to these methods provide representations of program parts (e.g., classes) as mathematical objects describing how program parts modify the stack and heap. However, these denotations are often not *abstract* enough, i.e., they differentiate between classes that have the same behavior. Banerjee and Naumann [8] presented a method to reason about whole-program equivalence in a Java subset. Under a notion of confinement for class tables, they prove equivalence between different implementations of a class by relating their (classical, fixpoint-based) denotations by simulations. In subsequent work [9], they use a discipline using assertions and ghost fields to specify invariants and heap encapsulation (by ownership techniques) and to deal with reentrant callbacks. Jeffrey and Rathke [10] give a fully abstract trace semantics based on an operational semantics for a Java subset with a package-like construct. Class types are always package-local and interfaces public. The package system is used to encapsulate objects (package declarations can contain object declarations) and thus delimit the boundaries of interactions, which makes it a lightweight library model. Furthermore, they do not consider inheritance, down-casting and cross-border instantiation. Using similar techniques, Steffen [11] and Ábrahám et al. [12] give a fully abstract trace semantics for a concurrent class-based language (without inheritance and subtyping). Similar as in [10], the runtime configurations of the operational semantics are modeled in a way that they syntactically capture what is part of the component under observation and what is part of the context, which makes the configurations rather non-standard (component and context each have their own stack and heap). The full abstraction proof strongly relies on the property that the traces can be decomposed into complementary traces (and recomposed) due to the duality of the component and context. Reasoning in terms of the trace-based semantics is not further explored in either work. In the setting of concurrent data structures, Gotsman and Yang [13] use traces and a most general client to check whether a library linearizes another. Filipovic et al. [14] use a trace abstraction to study whether sequential consistency or linearizability implies observational refinement.

Bisimulations were first used by Hennessy and Milner [15] to reason about concurrent programs. Sumii and Pierce used bisimulations which are sound and complete with respect to contextual equivalence in a language with dynamic sealing [16] and in a language with type abstraction and recursion [17]. Koutavas and Wand, building on their earlier work [18] and the work of Sumii and Pierce, used bisimulations to reason about the equivalence of *single* classes [19] in different Java subsets. The subset they considered includes inheritance and down-casting but neither interfaces nor accessibility of types. Showing equivalence between two class implementations in their setting amounts to constructing adequate relations (which are sound and complete w.r.t. contextual equivalence). These adequate relations are similar[2] to the specialized simulation relations outlined in Section 8.2 (the auxiliary relation R^l in their work to talk about related objects corresponds to our correspondence relation ρ). As a proof method, the authors provide conditions that are sufficient for proving adequacy of a given relation. In the setting of class libraries, deriving such conditions is a lot more complicated, as it is e.g. not statically clear which objects or method bodies need to be related (see Section 8.1).

We give a fully abstract trace-based semantics for class libraries of a *sequential* object-oriented programming language with the typical OO features, namely interfaces, classes, inheritance and subtyping. To model encapsulation aspects, we consider a package system which hides package-local types. As our semantics is strongly geared towards practical application, the language is a more faithful subset of Java than [10]. The main idea behind our fully abstract semantics is to combine characteristics from (1) denotational, trace-based semantics, i.e., the mental model of traces which characterize the behavior of a library in terms of its input and output, as well as (2) bisimulation approaches, i.e., by providing a strong link to a standard operational semantics which leads to a direct connection to Hoare-like program logics [20].

The idea behind our reasoning method is based on the states where control is outside of the library, which we call the *observable states*[3]. The (coupling) invariant that relates the configurations of both libraries must hold at corresponding observable states in the execution. As libraries are comprised of multiple classes, the observable states are, in contrast to [21–23], not statically bound to program points like start and end of methods.

Our approach is modular in the sense that we do not need knowledge about the contexts of the library. In contrast to Banerjee and Naumann [9], we consider multiple classes and we do not need a special *inv/own* discipline as described in [24], i.e., an explicit representation of when object invariants are known to hold. In our case, it is clear when the invariant must hold, namely in observable states which result directly from the language and module system. On one hand, we have a less parametric framework than Banerjee and Naumann as the notion of confinement (that results from the encapsulation offered by the module system, in our case Java packages) is fixed by the language semantics. On the other hand, our framework is more powerful as it is not tied to an external confinement discipline. In particular, our reasoning method is complete with respect to contextual equivalence. We allow for example different boundary objects to share their representation (e.g. a list with iterators), which is not possible in [9].

This paper is a completely reworked version of an earlier paper published at SBMF [25]. Beside a more streamlined notation and terminology, we include more examples, provide proofs for the full abstraction theorem and more details about proving backward compatibility, and give a detailed account of the related work.

[2]Whereas their proof goes by induction on the complexity of possible expressions (where by complexity we mean the number of steps it takes to reduce the expression to a value), our proof goes by induction on the length of the traces of the library with the most general context.

[3]We use the term *observable states* in allusion to the *visible states* based techniques [21, 22].

1.3. Outline

The remainder of this paper is structured as follows. Section 2 introduces the formalized Java dialect, called LPJAVA, for which we study backward compatibility. A prerequisite for backward compatibility is that all client code that compiles against the old version of a library also compiles against the new version. This property is known under the name of *source compatibility* and solely depends on the well-formedness conditions and type system of the language. To capture this aspect in a formal way, we present a type system for LPJAVA in Section 3 and formalize the notion of source compatibility in Section 4. Section 5 then gives the operational and trace semantics of LPJAVA. Section 6 introduces a formal definition of observable behavior and presents the full abstraction result. Section 7 shows how full abstraction can be proved by specialized simulation relations and Section 8 gives a proof method for reasoning about backward compatibility of libraries. Section 9 presents directions for future work and concludes.

Notations. We use the *overbar* notation $\overline{x}$ to denote a finite sequence and the *hat* notation $\widehat{x}$ to denote a finite set. The empty sequence and set are denoted by $\bullet$ and the concatenation of sequence $\overline{x}$ and $\overline{y}$ is denoted by $\overline{x} \cdot \overline{y}$. Concatenation is sometimes implicit by writing terms in juxtaposition. Single elements are implicitly treated as sequences/sets when needed. The function $\mathsf{last}(\ldots)$ returns the last element of a sequence. The expression $\mathcal{M}[x \mapsto y]$ yields the map $\mathcal{M}$ where the entry with key x is updated with the value y, or, if no such key exists, the entry is added. The empty map is denoted by $\varnothing$ and $\mathsf{dom}(\mathcal{M})$ and $\mathsf{rng}(\mathcal{M})$ denote the domain and range of the map $\mathcal{M}$. In order to improve readability, we often use the underscore place-holder _ instead of a free logical variable that occurs only once in the formula. Larger proofs are written in a hierarchically structured style as advocated by Lamport [26].

2. Formalization of LPJAVA

The formalized language considered in the following is a sequential object-oriented language called LPJAVA (*Lightweight Package Java*). It has the standard object-oriented features like classes, interfaces, inheritance, method calls, and mutable state. To allow for interesting scenarios of library evolution, we also consider the Java namespace mechanism to hide certain types in libraries, known under the name of a package system [27].

The syntax of LPJAVA is presented in Figure 4. We use the following meta-variables to represent elements of different syntactic categories: c denotes class names (incl. Object), i interface names, p package names (incl. lang), f field names, m method names and x local variable names (incl. this). For simplicity, we mix syntactic categories with names of their typical elements. A *package* (denoted by P) has a name and consists of a sequence of type declarations. We assume that packages are sealed (cf. [28], Sect. 2), meaning that once a package is defined no new class and interface definitions can be added to the package. Types are fully qualified by their package name. Classes and interfaces can be declared either package-local or public. Primitive data types (like bool, int, etc.) are not considered as they do not provide additional insight. For simplicity, all methods are assumed to be public and all fields to be private. We omit the public and private modifiers on methods and fields in the following. We assume that every class has a default constructor. Similar as for Java [27], the default constructor has the same access modifier as its class. LPJAVA also allows explicit casting, which leads to more distinguishing power from class contexts. The operator $(p.t)E_1$ err E_2 encodes both an *instanceof* and *cast* operator, i.e., it yields the value of E_2 if the value of E_1 cannot be cast to $p.t$. Sequencing of expressions can be done using the expression let $p.t\ x = E_1$ in E_2. To make the program text more readable, we allow to abbreviate the previous expression by E_1; E_2 if x does not appear in E_2.

To establish a precise terminology throughout the paper, we give formal definitions for what we consider as codebase, library implementation, program and program context.

Definition 2.1 (Codebase K,X,Y)**.** A *codebase* consists of a sequence of packages (i.e. $\overline{P}$) and is denoted by the meta-variables K, X or Y.

Definition 2.2 (Library implementation)**.** A codebase is called a *library implementation* if it satisfies all the well-formedness conditions of the language, i.e., well-formed type hierarchy, well-typedness of method bodies, etc. Well-formedness of a codebase X is formalized as $\vdash X$ in Figure 6.

Definition 2.3 (Program and program context)**.** A *program* is a codebase that has a main class with a main method and that satisfies all the well-formedness conditions of the language. If we join a codebase K (with a main method) and a library implementation X to form a program, we call K a *program context* of X.

To join two codebases into a larger codebase, we write them in juxtaposition (e.g. KX). In the following, we use K for codebases that take the role of program contexts and X and Y for codebases that take the role of library implementations. Note that library implementations are definition-complete, i.e., every type that is used in the library implementation must be declared in the library implementation.

3. Well-formedness and Typing

In this section, we introduce the notation and rules used to describe the well-formedness and typing conditions for LPJava. Most of the presented relations take the codebase X to be checked as an explicit parameter. Often this parameter is left implicit in language specifications. However, as we want to talk later on about different codebases when comparing them for compatibility, it is useful to make explicit which codebase we are referring to.

We denote by $\mathsf{uniquenames}_X$ that package declarations in a codebase X do not share the same package name. We further assume that class and interface names in each package are unique and field and method names declared in each type are unique. We define $\mathcal{P}_X$ as the set of package names for which there is a package declaration in X. We define $\mathcal{C}_X$ as the set of (qualified) class identifiers for which there is a declaration in X. Similarly, $\mathcal{I}_X$ represents the set of declared interfaces. We then define $\mathcal{C}_X^{\mathbf{Obj}} \overset{\text{def}}{=} \mathcal{C}_X \cup \{\mathsf{lang.Object}\}$ and the set of types $\mathcal{T}_X \overset{\text{def}}{=} \mathcal{C}_X^{\mathbf{Obj}} \cup \mathcal{I}_X \cup \{\bot\}$, where $\bot$ will be used to type the null expression and is called the *null type*. To simplify the presentation we use the meta-variable T to denote all kind of types ($T ::= p.c \mid p.i \mid \bot$). The public types of X are characterized by the predicate public_X. In particular, $\mathsf{public}_X(\mathsf{lang.Object})$ and $\mathsf{public}_X(\bot)$ hold for any codebase X under consideration.

The following symbols express relations between the types. For a codebase X, we define the direct subtype relation $<_X^{\mathbf{d}}$ as the least relation with the following properties. The relation contains $p_1.c_1 <_X^{\mathbf{d}} p_2.c_2$ if a class c_1 in package p_1 has an extds clause mentioning the class $p_2.c_2$. Similarly, it contains $p_1.i_1 <_X^{\mathbf{d}} p_2.i_2$ if an interface i_1 in package p_1 has an extds clause mentioning the interface $p_2.i_2$. If a class c_1 in package p_1 mentions an interface $p_2.i_2$ in its impls clause, the relation contains $p_1.c_1 <_X^{\mathbf{d}} p_2.i_2$. The direct subtype relation also contains $p.i <_X^{\mathbf{d}} \mathsf{lang.Object}$ for each interface type $p.i$ in $\mathcal{I}_X$ that has an empty extds clause. By $<_X$ we denote the transitive closure of $<_X^{\mathbf{d}} \cup \{(\bot, p.t) \mid p.t \in \mathcal{C}_X^{\mathbf{Obj}} \cup \mathcal{I}_X\}$, by $\leq_X$ the reflexive closure of $<_X$.

To describe that a type declaration provides methods or fields, we introduce the membership relation $\in_X$. We write $\langle f, p_0.t_0\rangle \in_X p.c$ to describe that a field f of type $p_0.t_0$ is declared in a class c of package p. In contrast to fields which are always private, methods are public and can thus be inherited. The membership symbol $\in_X$ then also captures the idea of considering all transitively inherited members. We write $\langle m, \overline{T}, T\rangle \in_X p.t$ to describe that a method m of signature $\overline{T}$ is member of the type $p.t$. The additional type T is used to denote in which supertype the method is implemented if the method is inherited. Method signatures are simply written as a sequence $\overline{T} \overset{\text{def}}{=} (p_0.t_0 \cdot \overline{p_1.t_1})$ where the first element $p_0.t_0$ of the sequence represents the return type of the method and $\overline{p_1.t_1}$ denote the parameter types. Method membership is defined inductively. We have two base cases (1) If a method m with signature $\overline{T}$ is declared in a class $p.c$ then $\langle m, \overline{T}, p.c\rangle \in_X p.c$. (2) If a method m with signature $\overline{T}$ is declared in an interface $p.i$ then $\langle m, \overline{T}, \bot\rangle \in_X p.i$. We use $\bot$ as a place-holder as methods in interfaces provide no implementation. The inductive steps are then given in the rules below. A method is inherited from a superclass if it is not overridden, i.e. there is no method with the same name and signature declared in the inheriting class (D-INH-METHOD-C). Methods from superinterfaces are always inherited (D-INH-METHOD-I).

$$\textsc{d-inh-method-c}\quad \frac{p_1.c_1 <_X^{\mathbf{d}} p_2.c_2 \qquad \langle m, \overline{T}, T\rangle \in_X p_2.c_2 \qquad \langle m, \overline{T}, p_1.c_1\rangle \notin_X p_1.c_1}{\langle m, \overline{T}, T\rangle \in_X p_1.c_1} \qquad \textsc{d-inh-method-i}\quad \frac{p_1.i_1 <_X^{\mathbf{d}} p_2.i_2 \qquad \langle m, \overline{T}, \bot\rangle \in_X p_2.i_2}{\langle m, \overline{T}, \bot\rangle \in_X p_1.i_1}$$

Using the presented notation, we formalize well-formedness of a codebase in Figure 6. The following judgments are used:

▹ $\vdash X$ denotes that the codebase X is well-formed, i.e., X is a library implementation.

$$
\begin{array}{rl}
P ::= & \text{package } p\ ;\ \overline{D} \\
D ::= & \text{public}^{?}\ \text{class } c \text{ extds } p.c \text{ impls } \overline{p.i}\ \{\ \overline{F}\ \overline{M^{\mathbf{c}}}\ \} \\
| & \text{public}^{?}\ \text{interface } i \text{ extds } \overline{p.i}\ \{\ \overline{M^{\mathbf{a}}}\ \} \\
F ::= & \text{private } p.t\ f\,; \\
M^{\mathbf{a}} ::= & \text{public } p.t\ m(\overline{p.t\ x})\ ; \\
M^{\mathbf{c}} ::= & \text{public } p.t\ m(\overline{p.t\ x})\ \{\ E\ \} \\
E ::= & x \mid \text{null} \mid \text{new } p.c \mid (p.t)E \text{ err } E \mid E.f \mid E.f = E \\
| & \text{let } p.t\ x = E \text{ in } E \mid E.m(\overline{E}) \mid (E == E\ ?\ E : E) \\
t ::= & c \mid i
\end{array}
$$

Figure 4: Syntax of LPJava. $^{?}$ denotes an optional item.

$$
\begin{array}{lr}
p_1.t_1 <^{\mathbf{d}}_X p_2.t_2 \rightarrow \text{acc}_X(p_2.t_2, p_1) & (\text{C1}_X) \\
\langle m, \overline{T_1}, \_\rangle \in_X T \wedge \langle m, \overline{T_2}, \_\rangle \in_X T \rightarrow \overline{T_1} = \overline{T_2} & (\text{C2}_X) \\
\langle m, \overline{T}, \_\rangle \in_X p_1.i_1 \wedge p_2.c_2 <^{\mathbf{d}}_X p_1.i_1 \rightarrow \langle m, \overline{T}, \_\rangle \in_X p_2.c_2 & \\
 & (\text{C3}_X) \\
\text{public}_X(T) \wedge \langle m, \overline{T}, \_\rangle \in_X T \rightarrow \text{public}_X(\overline{T}) & (\text{C4}_X)
\end{array}
$$

Figure 5: Context rules for a codebase X

$$
\textsc{T-Lib}\ \frac{\begin{array}{c}\text{uniquenames}_X \qquad <_X \text{ is acyclic} \\ \text{C1}_X\text{-C4}_X \qquad X = \overline{P} \qquad X \vdash \overline{P}\end{array}}{\vdash X}
\qquad
\textsc{T-Package}\ \frac{\begin{array}{c}P = \text{package } p;\ \overline{D} \\ p \neq \text{lang} \qquad X, p \vdash \overline{D} \\ \exists t : p.t \in \mathcal{C}_P \cup \mathcal{I}_P \wedge \text{public}_P(p.t)\end{array}}{X \vdash P}
$$

$$
\textsc{T-Method}\ \frac{\begin{array}{c}X, p.c \vdash p_0.t_0\ m(\overline{p_1.t_1\ x})\ ; \\ \Gamma = \varnothing[\text{this} \mapsto p.c][\overline{x} \mapsto \overline{p_1.t_1}] \\ X, p.c, \Gamma \vdash E :_{\leq} p_0.t_0\end{array}}{X, p.c \vdash p_0.t_0\ m(\overline{p_1.t_1\ x})\ \{\ E\ \}}
$$

$$
\textsc{T-Class}\ \frac{\begin{array}{c}\text{acc}_X(\overline{p_1.t_1}, p) \\ X, p.c \vdash \overline{M^{\mathbf{c}}}\end{array}}{X, p \vdash \ldots \text{class } c \ldots \{\overline{p_1.t_1\ f};\ \overline{M^{\mathbf{c}}}\}}
\qquad
\textsc{T-Intf}\ \frac{X, p.i \vdash \overline{M^{\mathbf{a}}}}{X, p \vdash \ldots \text{interface } i \ldots \{\overline{M^{\mathbf{a}}}\}}
\qquad
\textsc{T-MethSig}\ \frac{\begin{array}{c}(\text{this} \cdot \overline{x}) \text{ pairwise distinct} \\ \text{acc}_X(p_0.t_0 \cdot \overline{p_1.t_1}, p)\end{array}}{X, p.t \vdash p_0.t_0\ m(\overline{p_1.t_1\ x})\ ;}
$$

$$
\textsc{T-Null}\ \ X, p.c, \Gamma \vdash \text{null} : \bot
\qquad
\textsc{T-Var}\ \frac{\Gamma(x) = p_1.t_1}{X, p.c, \Gamma \vdash x : p_1.t_1}
\qquad
\textsc{T-New}\ \frac{\text{acc}_X(p_1.c_1, p)}{X, p.c, \Gamma \vdash \text{new } p_1.c_1() : p_1.c_1}
\qquad
\textsc{T-Get}\ \frac{\begin{array}{c}X, p.c, \Gamma \vdash E : p.c \\ \langle f, p_1.t_1\rangle \in_X p.c\end{array}}{X, p.c, \Gamma \vdash E.f : p_1.t_1}
$$

$$
\textsc{T-Set}\ \frac{\begin{array}{c}X, p.c, \Gamma \vdash E_1.f : p_1.t_1 \\ X, p.c, \Gamma \vdash E_2 :_{\leq} p_1.t_1\end{array}}{X, p.c, \Gamma \vdash E_1.f = E_2 : p_1.t_1}
\qquad
\textsc{T-Call}\ \frac{\begin{array}{c}X, p.c, \Gamma \vdash E : p_1.t_1 \\ \langle m, p_0.t_0 \cdot \overline{p_2.t_2}, \_\rangle \in_X p_1.t_1 \\ X, p.c, \Gamma \vdash \overline{E :_{\leq} p_2.t_2}\end{array}}{X, p.c, \Gamma \vdash E.m(\overline{E}) : p_0.t_0}
\qquad
\textsc{T-Let}\ \frac{\begin{array}{c}X, p.c, \Gamma \vdash E_1 :_{\leq} p_1.t_1 \\ X, p.c, \Gamma[x \mapsto p_1.t_1] \vdash E_2 : T \\ x \notin \text{dom}(\Gamma)\end{array}}{X, p.c, \Gamma \vdash \text{let } p_1.t_1\ x = E_1 \text{ in } E_2 : T}
$$

$$
\textsc{T-If}\ \frac{\begin{array}{c}X, p.c, \Gamma \vdash E_i : T_i \qquad \text{cmp}_X(T_1, T_2) \\ \text{cmp}_X(T_3, T_4) \qquad T = \text{max}_X(T_1, T_2)\end{array}}{X, p.c, \Gamma \vdash (E_1 == E_2\ ?\ E_3 : E_4) : T}
\qquad
\textsc{T-Cast}\ \frac{\begin{array}{c}\text{acc}_X(p_1.t_1, p) \qquad X, p.c, \Gamma \vdash E_i : T_i \\ \text{cmp}_X(p_1.t_1, T_1) \qquad T_2 \leq_X p_1.t_1\end{array}}{X, p.c, \Gamma \vdash (p_1.t_1)E_1 \text{ err } E_2 : p_1.t_1}
\qquad
\textsc{T-Sub}\ \frac{\begin{array}{c}X, p.c, \Gamma \vdash E : T \\ \text{acc}_X(p_1.t_1, p) \\ T \leq_X p_1.t_1\end{array}}{X, p.c, \Gamma \vdash E :_{\leq} p_1.t_1}
$$

Figure 6: Well-formedness of codebases and typing rules, using definitions from Figures 5 and 7

$$
\begin{array}{l}
\text{acc}_X(p.t, p') \overset{\text{def}}{=} \begin{cases} p.t \in \mathcal{T}_X \wedge \\ \quad (\text{public}_X(p.t) \vee p = p') \end{cases} \\
\text{cmp}_X(T_1, T_2) \overset{\text{def}}{=} T_1 \leq_X T_2 \vee T_2 \leq_X T_1 \\
\text{max}_X(T_1, T_2) \overset{\text{def}}{=} \begin{cases} T_2 & \text{if } T_1 \leq_X T_2 \\ T_1 & \text{if } T_2 \leq_X T_1 \end{cases}
\end{array}
$$

Figure 7: Additional typing definitions

$$
\begin{array}{lr}
\mathcal{P}_X = \mathcal{P}_Y & (\text{S1}_{X,Y}) \\
\text{public}_X(T) \rightarrow \text{public}_Y(T) & (\text{S2}_{X,Y}) \\
\text{public}_X(T) \rightarrow (\langle m, \overline{T}, \_\rangle \in_X T \leftrightarrow \langle m, \overline{T}, \_\rangle \in_Y T\ ) & (\text{S3}_{X,Y}) \\
\text{public}_X(T_1) \wedge \text{public}_X(T_2) \wedge T_1 \leq_X T_2 \rightarrow T_1 \leq_Y T_2 & (\text{S4}_{X,Y})
\end{array}
$$

Figure 8: Conditions to check source compatibility in LPJava

- ▹ $X \vdash P$ denotes that the package declaration P is well-formed in codebase X.
- ▹ $X, p \vdash D$ denotes that the type declaration D (class or interface) of package p is well-formed in codebase X.
- ▹ $X, p.t \vdash M^{\mathbf{a}}$ and $X, p.t \vdash M^{\mathbf{c}}$ denote that the methods $M^{\mathbf{a}}$ and $M^{\mathbf{c}}$ declared in type $p.t$ are well-formed in X.
- ▹ $X, p.c, \Gamma \vdash E : T$ denotes that the expression E in class $p.c$ has the type T under local variable typing Γ.
- ▹ $X, p.c, \Gamma \vdash E :_{\leq} p'.t'$ denotes that expression E in class $p.c$ has a type that is a subtype of $p'.t'$. The judgment is defined only for types different to the null-type $\perp$. We have introduced this additional judgement in order to reduce the amount of logical variables in the typing rules. Its definition is given by rule **T-Sub** in Figure 6.

A codebase X is well-formed (see rule **T-Lib** in Figure 6) if it satisfies the context conditions $\mathsf{C1}_X$-$\mathsf{C4}_X$ and each package declaration in it is well-formed. The context conditions for a codebase X are defined in Figure 5. Free logical variables in the conditions are universally quantified.

Condition $\mathsf{C1}_X$ states that types occurring in extds and impls clauses must be accessible. Accessibility, defined as $\mathsf{acc}_X(p.t, p')$ in Figure 7, guarantees that a type is defined and either public or part of the same package. Condition $\mathsf{C2}_X$ enforces the absence of method overloading. Condition $\mathsf{C3}_X$ ensures that a class implements all the methods of the interfaces that it implements. Condition $\mathsf{C4}_X$ states that the signature of (public) methods (defined or inherited) in public classes must only contain public types. The C# language specification [29] defines similar restrictions. In particular, [29, Sect. 10.5.4 on accessibility constraints] presents conditions that among others require parameter and return types to be at least as accessible as the method itself. These additional constraints lead to important properties; they ensure for example that public interfaces can always be implemented.

A package declaration is well-formed (**T-Package** in Figure 6) if each type declaration in it is well-formed (and so on). Field and method types must be accessible in the package in which they are used (**T-Class**, **T-MethSig** and **T-Method**). The typing of expressions is straightforward. Note that the rule **T-If** only allows expressions of comparable type (modeled using the predicate $\mathsf{cmp}_X(T_1, T_2)$ in Figure 7).

4. Source Compatibility

A prerequisite for a library implementation to be backward compatible with another one is that whenever the first library implementation joined with a program context yields a program, then the second library implementation joined with the same program context must also yield a program. This property, focusing solely on typing and not behavioral aspects, is called *source compatibility*. A more detailed overview of source compatibility and related work is given in [30], where we studied a type system to support interface evolution of class libraries by giving explicit interfaces to packages.

Definition 4.1 (Source compatibility)**.** A library implementation Y is *source compatible* with a library implementation X if for any codebase K: $\vdash KX$ implies $\vdash KY$.

The definition is not useful to compute that a library implementation Y is source compatible with X, because it quantifies over an infinite set of contexts. However, a set of checkable conditions that are necessary and sufficient for Y to be source compatible with X can be given. The conditions, which can be found in Figure 8, are described in the following. The package names occurring in X must exactly be those occurring in Y ($\mathsf{S1}_{X,Y}$). Every public type defined in X must appear in Y ($\mathsf{S2}_{X,Y}$). Note that this implies that public class (interface) types declared in X are public class (interface) types declared in Y, because the set of class and interface identifiers are disjoint in LPJava. For public types of X, every method which is part of the type (declared or inherited) in X must also have a method with the same signature in Y and vice versa ($\mathsf{S3}_{X,Y}$). Finally, the subtype hierarchy between public types of X must be preserved in Y ($\mathsf{S4}_{X,Y}$). We state that the conditions are necessary and sufficient in the following theorem. For a rationale we refer to the proofs.

Theorem 1 (Soundness and completeness of checkable conditions)**.** *A library implementation Y is source compatible with a library implementation X if and only if $\mathsf{S1}_{X,Y}$-$\mathsf{S4}_{X,Y}$ hold.*

Proof: By Lemmas A.1 and A.2 in Appendix A.1. □

5. Trace Characterization of Library Behavior

In this section, we enrich a standard operational semantics in such a way that the interactions between a library and its program contexts become explicit and call it the enhanced semantics (Section 5.1). Based on the enhanced semantics, we characterize the behavior of a library implementation X in terms of its possible interaction traces with program contexts. We first define the interaction traces of X with ordinary program contexts (Section 5.2). Then, we introduce nondeterministic expressions that allow for the definition of most general contexts which simulate all possible contexts (Section 5.3).

5.1. *Enhanced Operational Semantics*

The small-step operational semantics of LPJava, presented in the style of FJ [31], is given by the rules in Figure 11. Auxiliary functions are defined in Figure 14. The operational rules for a program that consists of a library implementation X and a program context K are based on a labelled small-step reduction judgement of the form $\mathcal{S} \overset{\gamma}{\rightsquigarrow} \mathcal{S}'$ with configurations $\mathcal{S} \overset{\text{def}}{=} KX, \mathcal{O}, \overline{\mathcal{F}}$. Labels record changes of control and are described in more detail in the next subsection. In order to generate the traces and realize the most general context, the configurations and rules are augmented with additional information (highlighted in Figures 9 and 11). However, this additional information does not change the standard operational behavior.

The syntax of configurations is given in Figure 9. The heap $\mathcal{O}$ is a map from object identifiers to heap entries and the stack is represented as a sequence of typed interaction frames $\overline{\mathcal{F}}$. This partitioning of the stack into interaction frames allows to mark parts of the stack as belonging to the program context or the library. The flag L ranges over $\{\mathsf{ctxt}, \mathsf{lib}\}$ and indicates whether entities belong to the context K or to the library implementation X. It is used in interaction frames to mark if the code (E or $\mathcal{E}$) that is part of this interaction frame originates from X or K. It is also used in heap entries to denote whether the object has been created by code of X or K. The flag V is used in heap entries to denote whether an object created in the context has been exposed to the library or vice versa. Objects are always created internally (see **R-New**) but can over time be exposed when references to them are passed from code of the library to code of the context or vice versa.

Interaction frames are associated with either the library implementation X or the context K. The topmost interaction frame contains an expression E and all other interaction frames contain an evaluation context $\mathcal{E}$. An evaluation context $\mathcal{E}$ (see [32]) is an expression with a *hole* $[\,]$ somewhere inside the expression. We write $\mathcal{E}[E]$ to mean that the hole in $\mathcal{E}$ is replaced by expression E. A hole in $\mathcal{E}$ can only appear at certain positions defined as follows:

$$\begin{array}{rl} \mathcal{E} ::= & [\,] \mid \mathcal{E}.f \mid \mathcal{E}.f = E \mid v.f = \mathcal{E} \mid \mathsf{let}\ p.t\ x = \mathcal{E}\ \mathsf{in}\ E \mid (p.t)\mathcal{E}\ \mathsf{err}\ E \mid \mathcal{E}.m(\overline{E}) \\ \mid & v.m(\overline{v} \cdot \mathcal{E} \cdot \overline{E}) \mid (\mathcal{E} == E\ ?\ E : E) \mid (v == \mathcal{E}\ ?\ E : E) \end{array}$$

We say that X *controls execution* if code of X is executed; otherwise K controls execution. The function $\mathsf{exec}(\mathcal{S})$ from Figure 14 is used to determine who controls execution. An interaction is a change of control (see **R-Call-Boundary** and **R-Return-Boundary**). Note that only interactions allocate or deallocate interaction frames, i.e., calls within the context or the library implementation are handled within the same interaction frame (see **R-Call-Intern**). We use the helper relation $\rightsquigarrow^{L}_{KX}$ to denote steps that are local to an evaluation context (see **R-Internal-Step**).

As described in Definition 2.3, a program context is a codebase that has a main class $p.c$ with a main method lang.Object main(), where the class $p.c$ is also called a *startup class*. It is executed by calling main. In the following we assume that the startup class has the name main.Main and is defined in the context K.

Definition 5.1 (Initial configuration $\mathcal{S}^{\mathsf{init}}_{KX}$)**.** The initial (startup) configuration $\mathcal{S}^{\mathsf{init}}_{KX}$ is defined as $KX, \mathcal{O}, \mathcal{F} \cdot \bullet$ where $\mathcal{O} \overset{\text{def}}{=} \emptyset[o \mapsto (\mathsf{internal}, \mathsf{ctxt}, \mathsf{main.Main}, \mathsf{initf}_{KX}(\mathsf{main.Main}))]$ and $\mathcal{F} \overset{\text{def}}{=} E[o/\mathsf{this}]_{\mathsf{ctxt}}{:}\mathsf{lang.Object}$, if E is the body of the main method and o is an arbitrarily chosen object identifier.

5.2. *Traces*

In the following, we consider (interaction) traces as sequences of labels $\overline{\gamma}$ which are generated by steps of the enhanced operational semantics. The syntax of labels is given in Figure 10. Interaction is considered from the viewpoint of the library. Input labels (marked by 📥) express a change of control from the context to the library; output labels (marked by 📤) express a change from the library to the context. The choice as to whether an input or output label is to be generated is done using the extra information L tagged to the interaction frames. Transitions which do not express a control flow change are marked as silent transitions with the label τ.

There are input and output labels for method invocation and return. The labels for method invocation and return include the parameter and result values together with their *abstracted types*. To compare traces of different library implementations in a way that is independent from standard program contexts, we abstract from types declared in the context (e.g., IntObs in Figure 3). Similarly, local types should not appear in the labels, because different library implementations might use different local types, i.e, renaming the ObsIter class in Figure 2 should not matter to program contexts. Types in labels are *abstracted* (see $\mathsf{typeabs}_{KX}(p.c)$ in Figure 14) to a representation which only preserves the information (1) which public supertypes of $p.c$ belong to the library and (2) which of their methods are not overridden by the context. In our example, the type of ObsIter objects is abstracted to $\langle\{\mathsf{lang.Object}, \mathrm{Iterator}\}, \{\mathrm{hasNext}, \mathrm{next}\}\rangle$ and the type of IntObs objects is abstracted to $\langle\{\mathsf{lang.Object}, \mathrm{Observer}\}, \bullet\rangle$. The reason for (1) is that these are the types of X that can be used in cast expressions in the context. Based on the label, it becomes thus clear which cast expressions will succeed and which not. The reason for (2) is that, based on the label, we know the methods that, if invoked with the object as receiver, lead to changes of control. As X defines a finite set of types (denoted by $\mathcal{T}_X$), there are only a finite set of abstracted types that can occur in traces with X. This set is denoted by $\mathcal{T}_X^\alpha$ and can be constructed from X.

To abstract from silent τ steps of a computation, we provide a large step version of the enhanced semantics (denoted $\xrightarrow{\overline{\gamma}}$) that is inductively defined by:

$$\textsc{L-Empty}\quad \frac{}{\mathcal{S} \xrightarrow{\bullet} \mathcal{S}} \qquad\qquad \textsc{L-Step}\quad \frac{\mathcal{S} \xrightarrow{\overline{\gamma}} \mathcal{S}' \qquad \overbrace{\mathcal{S}' \overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow}}^{i \text{ times},\ i\in\mathbb{N}} \mathcal{S}'' \qquad \mathcal{S}'' \overset{\gamma}{\rightsquigarrow} \mathcal{S}''' \qquad \gamma \neq \tau}{\mathcal{S} \xrightarrow{\overline{\gamma}\cdot\gamma} \mathcal{S}'''}$$

Every large step represents a finite number of τ steps followed by a non-τ step. Note that τ does not appear in labels of large steps. Large steps represent the execution of small steps up to the state right after the next non-τ label has been generated.

As can be seen from the transition rules, evaluation is deterministic (up to object naming). In order to deal in the traces with the non-deterministic choice of fresh object identifiers, we introduce (object) renamings.

Definition 5.2 (Renaming ρ)**.** A *renaming* is a bijective relation on object identifiers. We write ρ for such a relation.

We can then consider traces equivalent modulo a renaming. We have equivalent traces $\overline{\gamma_1} \equiv^\rho \overline{\gamma_2}$ iff the object identifiers appearing at the same positions in the traces are related under the renaming ρ and the types appearing at the same position are equal. In the following, we use the straightforward generalization of this definition of *equivalence modulo a renaming* ($\equiv^\rho$) to arbitrary syntactic terms.

Definition 5.3 (Equivalence on terms ($\equiv^\rho$))**.** Two syntactic terms are equivalent modulo a renaming (written $\equiv^\rho$) if the object identifiers appearing at the same positions in the terms are related under the renaming ρ and the remaining parts of the terms are syntactically equal. If we are not interested in a particular ρ, we omit it for brevity.

Our goal in this subsection was to characterize the behavior of a library implementation X in terms of its possible interaction traces with program contexts, which we achieve by the following definition.

Definition 5.4 (Trace semantics)**.** The *traces* of a library implementation X with a program context K are given by $\mathsf{traces}(KX) \overset{\text{def}}{=} \{\overline{\gamma} \mid \exists \mathcal{S} : \mathcal{S}_{KX}^{\mathbf{init}} \xrightarrow{\overline{\gamma}} \mathcal{S}\}$

Note that $\mathsf{traces}(KX)$ is closed w.r.t. renaming, i.e., if $\overline{\gamma_1} \in \mathsf{traces}(KX)$ and $\overline{\gamma_1} \equiv \overline{\gamma_2}$, then also $\overline{\gamma_2} \in \mathsf{traces}(KX)$. Furthermore, $\mathsf{traces}(KX)$ is prefix-closed and only refers to public types in X.

$E ::= \ldots \mid v$	extd. expressions
$v ::= o \mid \mathsf{null}$	value
$\mathcal{O} ::= o \mapsto (V, L, p.c, \mathcal{G})$	heap
$\mathcal{G} ::= (p.c, f) \mapsto v$	field mapping
$V ::= \mathsf{internal} \mid \mathsf{exposed}$	exposure flag
$L ::= \mathsf{ctxt} \mid \mathsf{lib}$	origin location
$\mathcal{F} ::= E_L{:}p.t \mid \mathcal{E}_L{:}p.t$	interaction frame

Figure 9: Semantic entities for LPJava, where $o \in$ object identifiers

$\gamma ::= \mu\boxed{\uparrow} \mid \mu\boxed{\downarrow} \mid \tau$	label
$\mu ::= \mathsf{call}\ o^\alpha.m(\overline{v^\alpha})$	call message
$\quad \mid\ \mathsf{rtrn}\ v^\alpha$	return message
$o^\alpha ::= o{:}T^\alpha$	abstracted object
$v^\alpha ::= o^\alpha \mid \mathsf{null}$	abstracted value
$T^\alpha ::= \langle \widehat{p.t}, \widehat{m} \rangle$	abstracted type

Figure 10: Syntax of traces, i.e., sequences of labels $\overline{\gamma}$

R-Internal-Step
$$\frac{\mathcal{O}, E \leadsto^{L}_{KX} \mathcal{O}', E'}{KX, \mathcal{O}, \mathcal{E}[E]_L{:}p.t \cdot \overline{\mathcal{F}} \overset{\tau}{\leadsto} KX, \mathcal{O}', \mathcal{E}[E']_L{:}p.t \cdot \overline{\mathcal{F}}}$$

R-New
$$\frac{o \notin \mathsf{dom}(\mathcal{O}) \qquad \mathcal{G} = \mathsf{initf}_{KX}(p.c) \qquad \mathcal{O}' = \mathcal{O}[o \mapsto (\mathsf{internal}, L, p.c, \mathcal{G})]}{\mathcal{O}, \mathsf{new}\ p.c \leadsto^{L}_{KX} \mathcal{O}', o}$$

R-Cast
$$\frac{E' \equiv \mathsf{if}\ \mathsf{type}_{\mathcal{O}}(v) \leq_{KX} p.t\ \mathsf{then}\ v\ \mathsf{else}\ E}{\mathcal{O}, (p.t)v\ \mathsf{err}\ E \leadsto^{L}_{KX} \mathcal{O}, E'}$$

R-Let
$$\mathcal{O}, \mathsf{let}\ p.t\ x = v\ \mathsf{in}\ E \leadsto^{L}_{KX} \mathcal{O}, E[v/x]$$

R-If
$$\frac{E' \equiv \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2}{\mathcal{O}, (v_1 == v_2\ ?\ E_1 : E_2) \leadsto^{L}_{KX} \mathcal{O}, E'}$$

R-Get
$$\frac{\mathcal{O}(o) = (\_, \_, p.c, \mathcal{G})}{\mathcal{O}, o.f \leadsto^{L}_{KX} \mathcal{O}, \mathcal{G}(p.c, f)}$$

R-Call-Intern
$$\frac{\mathsf{type}_{\mathcal{O}}(o) = p.c \qquad \langle m, \_, p_0.c_0 \rangle \in_{KX} p.c \qquad p_0.c_0 \in \mathcal{C}_{\mathsf{select}_{KX}(L)} \qquad \mathsf{body}_{KX}(p_0.c_0, m) = (\overline{x}, E)}{\mathcal{O}, o.m(\overline{v}) \leadsto^{L}_{KX} \mathcal{O}, E[o/\mathsf{this}, \overline{v}/\overline{x}]}$$

R-Set
$$\frac{\mathcal{O}(o) = (V, L', p.c, \mathcal{G}) \qquad \mathcal{O}' = \mathcal{O}[o \mapsto (V, L', p.c, \mathcal{G}')] \qquad \mathcal{G}' = \mathcal{G}[(p.c, f) \mapsto v]}{\mathcal{O}, o.f = v \leadsto^{L}_{KX} \mathcal{O}', v}$$

R-Call-Boundary
$$\frac{\begin{array}{c}\mathsf{type}_{\mathcal{O}}(o) = p.c \qquad \langle m, \_, p_0.c_0 \rangle \in_{KX} p.c \qquad p_0.c_0 \in \mathcal{C}_{\mathsf{select}_{KX}(\neg L)} \\ \mathsf{body}_{KX}(p_0.c_0, m) = (\overline{x}, E) \qquad \gamma = \mathsf{mcall}_{KX}(o, m, \overline{v}, \mathcal{O}, L) \\ \mathcal{O}' = \mathsf{expose}(o \cdot \overline{v}, \mathcal{O}) \qquad \mathcal{F}' = E[o/\mathsf{this}, \overline{v}/\overline{x}]_{\neg L}{:}p'.t'\end{array}}{KX, \mathcal{O}, \mathcal{E}[o.m(\overline{v})]_L{:}p.t \cdot \overline{\mathcal{F}} \overset{\gamma}{\leadsto} KX, \mathcal{O}', \mathcal{F}' \cdot \mathcal{E}_L{:}p.t \cdot \overline{\mathcal{F}}}$$

R-Return-Boundary
$$\frac{\begin{array}{c}\gamma = \mathsf{mrtrn}_{KX}(v, \mathcal{O}, L) \\ \mathcal{O}' = \mathsf{expose}(v, \mathcal{O})\end{array}}{KX, \mathcal{O}, v_L{:}p'.t' \cdot \mathcal{E}_{\neg L}{:}p.t \cdot \overline{\mathcal{F}} \overset{\gamma}{\leadsto} KX, \mathcal{O}', \mathcal{E}[v]_{\neg L}{:}p.t \cdot \overline{\mathcal{F}}}$$

Figure 11: Transition rules for the enhanced small-step semantics, using the helper judgement $\leadsto^{L}_{KX}$ for internal steps that are local to an evaluation context. Helper definitions are in Figure 14 on page 15.

$E ::= \ldots \mid \mathsf{nde}$

T-Nde
$$X, p.c, \Gamma \vdash \mathsf{nde} : \bot$$

MGC-Skip
$$\mathcal{O}, \mathsf{nde} \leadsto^{\mathsf{ctxt}}_{KX} \mathcal{O}, \mathsf{nde}$$

MGC-New
$$\frac{o \notin \mathsf{dom}(\mathcal{O}) \qquad p.c \in \mathcal{T}_{KX} \qquad \mathsf{public}_{KX}(p.c)}{\mathcal{O}, \mathsf{nde} \leadsto^{\mathsf{ctxt}}_{KX} \mathcal{O}[o \mapsto (\mathsf{internal}, \mathsf{ctxt}, p.c, \mathsf{initf}_{KX}(p.c))], \mathsf{nde}}$$

MGC-Prepare-Call
$$\frac{\begin{array}{c}o \cdot \overline{v} \subseteq \mathsf{visible}(\mathcal{O}, \mathsf{ctxt}) \cup \{\mathsf{null}\} \\ \mathsf{type}_{\mathcal{O}}(o) = p.c \qquad \langle m, p_1.t_1 \cdot \overline{p_1.t_1}, p_0.c_0 \rangle \in_{KX} p.c \\ p_0.c_0 \in \mathcal{C}_X \qquad \mathsf{type}_{\mathcal{O}}(v) \leq_{KX} p_1.t_1 \qquad x\ \mathsf{fresh}\end{array}}{\mathcal{O}, \mathsf{nde} \leadsto^{\mathsf{ctxt}}_{KX} \mathcal{O}, \mathsf{let}\ \mathsf{lang.Object}\ x = o.m(\overline{v})\ \mathsf{in}\ \mathsf{nde}}$$

MGC-Prepare-Return
$$\frac{\begin{array}{c}v \in \mathsf{visible}(\mathcal{O}, \mathsf{ctxt}) \cup \{\mathsf{null}\} \\ \mathsf{type}_{\mathcal{O}}(v) \leq_{KX} p.t\end{array}}{KX, \mathcal{O}, \mathsf{nde}_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}} \overset{\tau}{\leadsto} KX, \mathcal{O}, v_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}}}$$

Figure 12: Syntax extension, typing and transition rules for the most general context

$E ::= \ldots \mid \mathsf{success}$

$\gamma ::= \ldots \mid \mathsf{succ}$

T-Success
$$X, p.c, \Gamma \vdash \mathsf{success} : \bot$$

R-Success
$$KX, \mathcal{O}, \mathcal{E}[\mathsf{success}]_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}} \overset{\mathsf{succ}}{\leadsto} KX, \mathcal{O}, \mathsf{null}_{\mathsf{ctxt}}{:}\bot$$

MGC-Prepare-Success
$$\mathcal{O}, \mathsf{nde} \leadsto^{\mathsf{ctxt}}_{KX} \mathcal{O}, \mathsf{success}$$

Figure 13: Observability for LPJava

5.3. *Most General Context*

Although the traces abstract from types and steps in the context, we still have to consider the traces of all possible program contexts in order to describe the full behavior of a library. This issue is addressed in this section by constructing a most general context $\mathsf{mgc}(X)$ based on the library implementation X that enables all possible interactions that X can engage in. The context $\mathsf{mgc}(X)$ represents exactly all contexts that X can have. Compared to a standard program context, $\mathsf{mgc}(X)$ abstracts over types, objects, and operational steps. To represent $\mathsf{mgc}(X)$, we extend LPJava by nondeterministic expressions (nde) and corresponding reduction rules (Figure 12). The nde expression has no influence on source compatibility as it is typed in the same way as null (**T-Nde**). Nondeterministic expressions are only allowed in most general contexts. Reducing a non-deterministic expression can lead to the creation of new objects of accessible types (**MGC-New**), a change in control with a well-formed method call or return using **R-Call-Boundary** or **R-Return-Boundary**. The rules **R-Call-Boundary** and **R-Return-Boundary** rely on the extra information V and L tagged to objects to determine what objects are available to the context ($\mathsf{visible}(\mathcal{O}, \mathsf{ctxt})$). Furthermore the rule **R-Return-Boundary** relies on the extra type information tagged to the interaction frame to choose a type-correct return value.

Construction of $\mathsf{mgc}(X)$. The most general context of X is constructed from the library implementation X which we denote by the construction function $\mathsf{mgc}(X)$. Let $p_{\mathbf{mgc}}$ be a package name not occurring in X. For each abstracted type $T^{\alpha} = \langle \widehat{p.t}, \widehat{m} \rangle \in \mathcal{T}_X^{\alpha}$, we construct a class of the following form in the package $p_{\mathbf{mgc}}$:

public class c extds $p_0.c_0$ impls $\overline{p.i}$ { $\widehat{M^{\mathsf{c}}}$ }

where (1) c is a class name that is unique for each abstracted type T^{α}, (2) $p_0.c_0$ is the smallest class in $\widehat{p.t}$, (3) $\overline{p.i}$ are the interfaces in $\widehat{p.t}$, (4) $\widehat{M^{\mathsf{c}}} \overset{\text{def}}{=} \{p_0.t_0\ m(\overline{p_1.t_1\ x})\ \{\ \mathsf{nde}\ \} \mid \langle m, p_0.t_0 \cdot \overline{p_1.t_1}, \_ \rangle \in_{KX} p.t \wedge p.t \in \widehat{p.t} \wedge m \notin \widehat{m}\}$. The idea behind this construction is that abstracting the type of the constructed class yields the abstracted type from which the class was constructed (i.e. $\mathsf{typeabs}_{\mathsf{mgc}(X)X}(p_{\mathbf{mgc}}.c) = T^{\alpha}$). For the Cell library implementations in Figure 1, the most general context would consist of four classes subclassing the Cell class, overriding either none or one of the methods get and set or both, and a class subclassing Object with no methods (if the Object class has no methods either).

The most general context $\mathsf{mgc}(X)$ also has an additional class Main, which is the startup class of $\mathsf{mgc}(X)X$:

```
package main; public class Main { lang.Object main() { nde } }
```

We confirm that $\mathsf{mgc}(X)X$ is a program (i.e. well-formed).

Definition 5.5 (Deterministic and most general contexts)**.** In order to distinguish standard program contexts of earlier sections from most general (program) contexts, we call the previous ones *deterministic* program contexts. Program contexts then simply subsume both deterministic and most general program contexts.

6. Full Abstraction

We present two notions of backward compatibility in Section 6.1, a classical one (called *contextual compatibility*) and a trace-based one, and state our main theorem, namely that both notions of compatibility are equivalent. In Section 6.2, we present the principal lemmas needed to prove both definitions equivalent.

6.1. *Observable Behavior*

Two program parts are usually considered equivalent if there is no context that is able to distinguish them. As this comparison involves running the program with all possible *contexts*, this definition is called *contextual compatibility*. This means that the context plays the role of an observer. A standard way to compare two program parts is to use termination behavior [33] or reachability of a certain state [34] (e.g. a pre-determined program point [11, 35]) as the observation result.

In this paper, we use as observation result whether a particular expression was reached in the program context. A simple way to realize this in our formalized language is to introduce a special expression success which we add to our surface syntax of LPJava in Figure 13. The expression has no influence on source compatibility as it is typed in the same way as null (**T-Success**). If, during a program run, we reach now such an expression (**R-Success**), then we stop the running program (as we are not interested in the further execution of the program, we only want to make the observation) and the special label succ is emitted to represent the observation in the trace. We also adapt the MGC such that it is also able to do observations (**MGC-Prepare-Success**). We put no restrictions on how often the success expression is used in programs. It is, however, reasonable to only allow it in the program context (so that success cannot be faked by the library) and possibly only once if we want to achieve the most distinguishability. We introduce symbols to denote whether a program run leads to an observation or not. Note that the success expression is reached in the program run if and only if the label succ occurs in the trace.

Definition 6.1 (Successful (✓) and unsuccessful (✗) programs)**.** We write $\mathcal{S}$✓ if there exists a configuration $\mathcal{S}'$ such that $\mathcal{S} \xrightarrow{\overline{\gamma}} \mathcal{S}'$ and $\mathsf{last}(\overline{\gamma}) = \mathsf{succ}$. We write $\mathcal{S}$✗ in the other cases.

We use ✓ and ✗ only for programs with deterministic program contexts. Using the formalized notion of observation, we can then define the standard notion of contextual compatibility, namely that any program context that can make an observation with the first library implementation must be able to do the observation with the second library implementation.

Definition 6.2 (Contextual compatibility)**.** A library implementation Y is *contextually compatible* with a library implementation X if Y is source compatible with X and for any deterministic program context K of X: $\mathcal{S}^{\mathbf{init}}_{KX}$✓ implies $\mathcal{S}^{\mathbf{init}}_{KY}$✓.

The definition of contextual compatibility quantifies over all possible program contexts and, as outlined for the challenges in the introduction, cannot be used in general for proving that two library implementations are compatible. However, using the definitions of traces and most general context, we can give the denotation of a library implementation X as the set of traces generated by the most general context with X. Note that this definition solely depends on X.

Definition 6.3 (Trace behavior of a library implementation)**.** The trace-based behavior of a library implementation X is defined as $\mathsf{traces}(\mathsf{mgc}(X)X)$.

Based on this definition, we can give a different characterization of backward compatibility which we call *trace compatibility*.

Definition 6.4 (Trace compatibility)**.** A library implementation Y is *trace compatible* with a library implementation X if Y is source compatible with X and $\mathsf{traces}(\mathsf{mgc}(X)X) \subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$.

We cannot simply state trace inclusion, as Y may have more public types than X (see Definition 4.1 and $\mathsf{S2}_{X,Y}$). Our solution is to abstract from these additional types in the traces with the function $\mathsf{abs}_X(T^\alpha)$ defined in Figure 14. Finally we can state our main theorem, namely that the compatibility notions of Definitions 6.2 and 6.4 coincide. This means that we can use the definition of trace compatibility instead of the one of contextual compatibility in order to study backward compatibility of class libraries.

Theorem 2 (Full abstraction)**.** *Consider two library implementations X and Y. Then Y is trace compatible with X iff Y is contextually compatible with X.*

Proof: The proof relies on the main lemmas in Section 6.2 and is given in Appendix A.2. □

Note that the definitions and properties stated so far can directly be transferred to a setting which studies the equivalence of library implementations, as equivalence is just compatibility in both directions.

Definition 6.5 (Source, contextual and trace equivalence)**.** Library implementations X and Y are ψ *equivalent*, $\psi \in \{$source, contextually, trace$\}$, if X is ψ compatible with Y and Y is ψ compatible with X.

$$\mathsf{select}_{KX}(L) \overset{\text{def}}{=} \begin{cases} K & \text{if } L = \mathsf{ctxt} \\ X & \text{if } L = \mathsf{lib} \end{cases}$$

$$\mathsf{from}(L) \overset{\text{def}}{=} \begin{cases} \boxed{\downarrow} & \text{if } L = \mathsf{ctxt} \\ \boxed{\uparrow} & \text{if } L = \mathsf{lib} \end{cases}$$

$$\mathsf{type}_{\mathcal{O}}(v) \overset{\text{def}}{=} \begin{cases} \bot & \text{if } v = \mathsf{null} \\ p.c & \text{if } \mathcal{O}(v) = (\_,\_,p.c,\_) \end{cases}$$

$$\begin{array}{ll}
\neg L & \overset{\text{def}}{=} L' \text{ where } L \neq L' \text{ (similar for } \neg V) \\
\mathsf{exec}(KX,\mathcal{O},\overline{\mathcal{F}}) & \overset{\text{def}}{=} L \quad \text{if } \overline{\mathcal{F}} = E_L{:}p.t \cdot \overline{\mathcal{F}'} \\
\mathsf{filter}(\mathcal{O},V) & \overset{\text{def}}{=} \{o \in \mathcal{O} \mid \mathcal{O}(o) = (V,\_,\_,\_)\} \\
\mathsf{filter}(\mathcal{O},L) & \overset{\text{def}}{=} \{o \in \mathcal{O} \mid \mathcal{O}(o) = (\_,L,\_,\_)\} \\
\mathsf{filter}(\mathcal{O},V,L) & \overset{\text{def}}{=} \mathsf{filter}(\mathcal{O},V) \cap \mathsf{filter}(\mathcal{O},L) \\
\mathsf{visible}(\mathcal{O},L) & \overset{\text{def}}{=} \mathsf{filter}(\mathcal{O},\mathsf{exposed}) \cup \mathsf{filter}(\mathcal{O},\mathsf{internal},L)
\end{array}$$

$$\begin{array}{ll}
\mathsf{expose}(v,\mathcal{O}) & \overset{\text{def}}{=} \begin{cases} \mathcal{O} & \text{if } v = \mathsf{null} \\ \mathcal{O}[o \mapsto (\mathsf{exposed},L,p.c,\mathcal{G})] & \text{if } v = o \wedge \mathcal{O}(o) = (\_,L,p.c,\mathcal{G}) \end{cases} \\
\mathsf{mcall}_{KX}(o,m,\overline{v},\mathcal{O},L) & \overset{\text{def}}{=} \mathsf{call}\ \mathsf{valabs}_{KX}(o,\mathcal{O}).m(\mathsf{valabs}_{KX}(\overline{v},\mathcal{O}))\ \mathsf{from}(L) \\
\mathsf{mrtrn}_{KX}(v,\mathcal{O},L) & \overset{\text{def}}{=} \mathsf{rtrn}\ \mathsf{valabs}_{KX}(v,\mathcal{O})\ \mathsf{from}(L) \\
\mathsf{valabs}_{KX}(v,\mathcal{O}) & \overset{\text{def}}{=} \begin{cases} \mathsf{null} & \text{if } v = \mathsf{null} \\ o{:}\mathsf{typeabs}_{KX}(\mathsf{type}_{\mathcal{O}}(o)) & \text{if } v = o \end{cases} \\
\mathsf{typeabs}_{KX}(p.c) & \overset{\text{def}}{=} \langle \widehat{p.t}, \widehat{m} \rangle \text{ where } \widehat{p.t} \overset{\text{def}}{=} \{p_0.t_0 \mid p_0.t_0 \in \mathcal{T}_X \wedge p.c \leq_{KX} p_0.t_0 \wedge \mathsf{public}_X(p_0.t_0)\} \\
& \qquad \text{and } \widehat{m} \overset{\text{def}}{=} \{m \mid \langle m,\_,p_0.c_0 \rangle \in_{KX} p.c \wedge p_0.c_0 \in \mathcal{C}_X\} \\
\mathsf{abs}_X(\langle \widehat{p.t}, \widehat{m} \rangle) & \overset{\text{def}}{=} \langle \widehat{p.t}', \widehat{m}' \rangle \text{ where } \widehat{p.t}' \overset{\text{def}}{=} \widehat{p.t} \cap \{p.t \mid p.t \in \mathcal{T}_X \wedge \mathsf{public}_X(p.t)\} \\
& \qquad \text{and } \widehat{m}' \overset{\text{def}}{=} \widehat{m} \cap \{m \mid \langle m,\_,\_ \rangle \in_X p.t \wedge p.t \in \widehat{p.t}'\} \\
\mathsf{body}_{KX}(p.c,m) & \overset{\text{def}}{=} (\overline{x},E) \text{ where } \overline{x} \text{ are the formal parameters} \\
& \qquad \text{and } E \text{ is the method body of the method } m \text{ defined in } p.c \\
\mathsf{initf}_{KX}(p.c) & \overset{\text{def}}{=} \{(p_0.c_0,f) \mapsto \mathsf{null} \mid \langle f,\_ \rangle \in_{KX} p_0.c_0 \wedge p.c \leq_{KX} p_0.c_0\} \\
\mathsf{stackabs}_L(\overline{\mathcal{F}}) & \overset{\text{def}}{=} \begin{cases} \bullet & \text{if } \overline{\mathcal{F}} = \bullet \\ \mathcal{F} \cdot \mathsf{stackabs}_L(\overline{\mathcal{F}}') & \text{if } \overline{\mathcal{F}} = \mathcal{F} \cdot \overline{\mathcal{F}}' \text{ and } \mathcal{F} = \_{}_L{:}p.t \\ \mathsf{stackabs}_L(\overline{\mathcal{F}}') & \text{if } \overline{\mathcal{F}} = \mathcal{F} \cdot \overline{\mathcal{F}}' \text{ and } \mathcal{F} = \_{}_{\neg L}{:}p.t \end{cases} \\
\mathsf{fields}^L_{KX}(\mathcal{G}) & \overset{\text{def}}{=} \{(p.c,f) \mapsto v \mid ((p.c,f) \mapsto v) \in \mathcal{G} \text{ and } p.c \in \mathcal{C}_{\mathsf{select}_{KX}(L)}\} \\
\mathsf{objectrefs}(\_) & \overset{\text{def}}{=} \text{yields all object identifiers contained in the syntactic element } \_
\end{array}$$

Figure 14: Helper definitions

6.2. Trace Properties

The following lemmas reveal the core properties of the trace semantics and form the constituents needed to prove full abstraction. Each of these lemmas are proven using specialized simulation relations. The simulation relations are presented in more detail in Section 7. The first two lemmas show that libraries and contexts compute the next label only based on the trace history, i.e., that the trace contains all relevant information. The proof of both lemmas is based on the specialized simulation relations $\preccurlyeq_{\mathsf{lib}}$ and $\preccurlyeq_{\mathsf{ctxt}}$. They relate runtime configurations of programs which have the same library implementation ($\preccurlyeq_{\mathsf{lib}}$) or program context ($\preccurlyeq_{\mathsf{ctxt}}$).

Lemma 6.1 (Library independency). *Consider two program contexts K_1 and K_2 for X such that $\overline{\gamma} \in \mathsf{traces}(K_1X)$ and $\overline{\gamma} \in \mathsf{traces}(K_2X)$ and $\mathsf{last}(\overline{\gamma}) = \mu\boxed{\downarrow}$. Then $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(K_1X)$ implies $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(K_2X)$.*

Proof: The initial states of K_1X and K_2X are related by $\preccurlyeq_{\mathsf{lib}}$ due to Lemma 7.3. By Corollary 7.10, the states right after the trace $\overline{\gamma}$ are related by $\preccurlyeq_{\mathsf{lib}}$ as well. By Lemma 7.8, the library implementations then give similar responses. □

Lemma 6.2 (Context independency). *Let Y be source compatible with X, K be a program context for X and Y, and $\overline{\gamma} \in \mathsf{traces}(KX)$ and $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$ and $\overline{\gamma} = \bullet$ or $\mathsf{last}(\overline{\gamma}) = \mu\boxed{\uparrow}$. Then, $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(KX)$ implies $\overline{\gamma} \cdot \gamma \in \mathsf{abs}_X(\mathsf{traces}(KY))$.*

Proof: Similar to the proof of the previous lemma by the relation $\preccurlyeq_{\mathsf{ctxt}}$, Lemmas 7.4 and 7.8 and Corollary 7.10. □

The following two lemmas state that the most general context for a library implementation X simulates exactly all possible contexts for X. The proof relies again on two specialized simulation relations. This time, however, we need stronger relations that do not only relate the library parts of the configurations, but also relate the most general context to the deterministic program context (denoted by $\lll$ for Lemma 6.3 and $\ggg$ for Lemma 6.4).

Lemma 6.3 (MGC abstraction is sound). *Let K be a deterministic program context of library implementation X. Then,* $\mathsf{traces}(KX) \subseteq \mathsf{traces}(\mathsf{mgc}(X)X)$.

PROOF: By using the simulation relation $\preccurlyeq_{\mathsf{lib}} \cap \lll$ between the configurations of a run of KX and the configurations of a run of $\mathsf{mgc}(X)X$ (Lemmas 7.11 and 7.12). □

Lemma 6.4 (MGC abstraction is complete). *Let X be a library implementation and $\overline{\gamma} \in \mathsf{traces}(\mathsf{mgc}(X)X)$. Then, there is a deterministic program context K of X with $\overline{\gamma} \in \mathsf{traces}(KX)$.*

PROOF: By constructing a deterministic program context from the trace (see Section 7.4.1) and using the specialized simulation relation $\preccurlyeq_{\mathsf{lib}} \cap \ggg$ in a similar way as for the previous lemma (Lemmas 7.13 and 7.14). □

The following lemma confirms the intuition that program contexts of X may only be able to explore parts of the behavior of Y.

Lemma 6.5 (MGC weakening). *Let Y be source compatible with X. Then* $\mathsf{traces}(\mathsf{mgc}(X)Y) \subseteq \mathsf{traces}(\mathsf{mgc}(Y)Y)$.

PROOF: Follows directly from the construction of $\mathsf{mgc}(X)$ and $\mathsf{mgc}(Y)$ as Y is source compatible with X. It can be shown that $\mathsf{mgc}(Y)$ contains all the classes of $\mathsf{mgc}(X)$ and steps of $\mathsf{mgc}(X)$ can be simulated by $\mathsf{mgc}(Y)$. □

Using the above lemma, trace compatibility can be restated in a form where it relies only on the most general context of X, namely $\mathsf{traces}(\mathsf{mgc}(X)X) \subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$. This property is directly used in the following lemma, which states that whenever we have two library implementations that respond in a different way when run with the most general context (i.e. generate different output labels), we can construct a deterministic program context that can observe this difference.

Lemma 6.6 (Differentiating context). *Let X and Y be source compatible library implementations such that $\overline{\gamma}_1 \cdot \gamma_1 \in$ $\mathsf{traces}(\mathsf{mgc}(X)X)$ and $\overline{\gamma}_2 \cdot \gamma_2 \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$ and $\mathsf{last}(\overline{\gamma}_1) = \mu$⍗ and $\overline{\gamma}_1 \equiv \overline{\gamma}_2$ but $\overline{\gamma}_1 \cdot \gamma_1 \not\equiv \overline{\gamma}_2 \cdot \gamma_2$. Then there is a deterministic program context K such that $\mathcal{S}^{\mathbf{init}}_{KX}$✓ and $\mathcal{S}^{\mathbf{init}}_{KY}$✗.*

PROOF: By adapting the construction from Lemma 6.4 (see Section 7.4.3). □

A last property formally confirms the intuition that observations can always be made when control of execution is outside the library.

Lemma 6.7 (MGC can always succeed). *Let X be a library implementation and $\overline{\gamma} \in \mathsf{traces}(\mathsf{mgc}(X)X)$ such that $\overline{\gamma} = \bullet$ or $\mathsf{last}(\overline{\gamma}) = \mu$⍐. Then $\overline{\gamma} \cdot \mathsf{succ} \in \mathsf{traces}(\mathsf{mgc}(X)X)$.*

PROOF: Follows directly from the construction of the MGC and the operational rule **MGC-PREPARE-SUCCESS**. □

7. Specialized Simulations For Proving Full Abstraction

In this section, we present the specialized simulation relations which are used to prove the main lemmas from Section 6.2. Before we can relate two configurations, we define in Section 7.1 a few well-formedness properties that the configurations to be related must satisfy. We then define the relations over well-formed runtime configurations in Sections 7.2 to 7.4. In Section 7.2 we give the relations $\preccurlyeq_{\mathsf{lib}}$ and $\preccurlyeq_{\mathsf{ctxt}}$ to prove Lemmas 6.1 and 6.2. In Section 7.3 we give the relation $\lll$ to prove Lemma 6.3 and in Section 7.4 we give the relation $\ggg$ to prove Lemma 6.4.

7.1. Well-formed Runtime Configurations

Well-formedness conditions on runtime configurations tell us about the invariants that hold during program runs (i.e. how we expect runtime configurations to look like). For example, absence of dangling pointers or well-typedness of references are standard well-formedness conditions. Before giving the definition of well-formed runtime configuration, we first present a few helper functions (formally defined in Figure 14). The function $\mathsf{stackabs}_L(\overline{\mathcal{F}})$ yields all the L-tagged interaction frames of $\overline{\mathcal{F}}$ and $\mathsf{fields}^L_{KX}(\mathcal{G})$ restricts $\mathcal{G}$ to fields that are defined in classes of L. The function $\mathsf{filter}(\mathcal{O},L)$ returns all object identifiers of objects in $\mathcal{O}$ that are tagged as L (similar for V). The function $\mathsf{visible}(\mathcal{O},L)$ returns the object identifiers of all exposed objects and objects internal to L. The function $\mathsf{objectrefs}(\dots)$ yields all object identifiers contained in a syntactic element.

Definition 7.1 (Well-formed runtime configuration). A runtime configuration $\mathcal{S} = KX,\mathcal{O},\overline{\mathcal{F}}$ is well-formed if

- ▹ $\mathcal{S}$ is well-typed (standard definition, not detailed further here)
- ▹ Top of stack $\overline{\mathcal{F}}$ is an expression of the form $\mathcal{E}[E]$, the rest are evaluation contexts of the form $\mathcal{E}$
- ▹ Stack frames in $\overline{\mathcal{F}}$ are alternatively from lib and from ctxt and the lowest stack frame is from ctxt
- ▹ Store consistency: $\mathsf{objectrefs}(\mathsf{rng}(\mathcal{O})) \subseteq \mathsf{dom}(\mathcal{O})$
- ▹ Stack consistency and separation: $\forall L : \mathsf{objectrefs}(\mathsf{stackabs}_L(\overline{\mathcal{F}})) \subseteq \mathsf{visible}(\mathcal{O},L)$
- ▹ Only L-visible objects can be accessed from L-visible objects: $\forall o \in \mathsf{visible}(\mathcal{O},L)$ with $\mathcal{O}(o) = (\_,\_,p.c,\mathcal{G})$ we have $\mathsf{rng}(\mathsf{fields}^L_{KX}(\mathcal{G})) \subseteq \mathsf{visible}(\mathcal{O},L)$
- ▹ Objects created by code of X are of a type of X: $\forall o \in \mathsf{filter}(\mathcal{O},\mathsf{lib}) : \mathsf{type}_{\mathcal{O}}(o) \in \mathcal{C}^{\mathsf{Obj}}_X$
- ▹ Internal objects of K have their X fields null: $\forall(\mathsf{internal},\mathsf{ctxt},p.c,\mathcal{G}) \in \mathsf{rng}(\mathcal{O}) : \mathsf{rng}(\mathsf{fields}^{\mathsf{lib}}_{KX}(\mathcal{G})) = \{\mathsf{null}\}$

In a similar fashion to the preservation lemma in type soundness proofs [32], we state in the following that runtime configurations of programs are always well-formed. Initial program states are well-formed (Lemma 7.1) and well-formedness is preserved by small operational steps (Lemma 7.2).

Lemma 7.1 (Initial state is well-formed). *The initial state $\mathcal{S}^{\mathsf{init}}_{KX}$ for a program KX is well-formed.*

Proof: Trivial. □

Lemma 7.2 (Preservation of well-formedness). *Consider a well-formed configuration $\mathcal{S}$. If $\mathcal{S} \overset{\gamma}{\rightsquigarrow} \mathcal{S}'$, then $\mathcal{S}'$ is well-formed as well.*

Proof: By case distinction on operational rule used. □

7.2. Context and Library Independency

The preorder relations $\preccurlyeq_{\mathsf{lib}}$ and $\preccurlyeq_{\mathsf{ctxt}}$ relate two well-formed runtime configurations if their lib or ctxt part is similar. This allows us for example to relate runtime configurations when programs only differ either in the context or library as in Lemmas 6.1 to 6.4. We give their definition in the following. We show in Section 7.2.1 how these preorder relations are affected by small operational steps and show in Section 7.2.2 that they have simulation properties on the large step relations.

We present the definitions of the relations $\preccurlyeq_{\mathsf{lib}}$ and $\preccurlyeq_{\mathsf{ctxt}}$ in a single definition $\preccurlyeq_L$, and rely on helper functions that are given in Figure 14. An informal explanation of the definition can be found below.

Definition 7.2 (Preorder relations $\preccurlyeq^{\rho}_L$). Consider two well-formed configurations of the form $\mathcal{S}_1 \overset{\mathrm{def}}{=} K_1X_1,\mathcal{O}_1,\overline{\mathcal{F}_1}$ and $\mathcal{S}_2 \overset{\mathrm{def}}{=} K_2X_2,\mathcal{O}_2,\overline{\mathcal{F}_2}$ such that X_2 is source compatible with X_1. We write $\mathcal{S}_1 \preccurlyeq^{\rho_e}_L \mathcal{S}_2$ if ρ_e is a renaming from $\mathsf{filter}(\mathcal{O}_1,\mathsf{exposed})$ to $\mathsf{filter}(\mathcal{O}_2,\mathsf{exposed})$ and there is a renaming ρ_i from $\mathsf{filter}(\mathcal{O}_1,\mathsf{internal},L)$ to $\mathsf{filter}(\mathcal{O}_2,\mathsf{internal},L)$ and $\rho = \rho_e \cup \rho_i$ such that

- ▹ if $L = \mathsf{ctxt}$ then $K_1 = K_2$ else $X_1 = X_2$
- ▹ $\mathsf{exec}(\mathcal{S}_1) = \mathsf{exec}(\mathcal{S}_2)$
- ▹ $\mathsf{stackabs}_L(\overline{\mathcal{F}_1}) \equiv^{\rho} \mathsf{stackabs}_L(\overline{\mathcal{F}_2})$
- ▹ If $o_1 \equiv^{\rho} o_2$ with $\mathcal{O}_1(o_1) = (V_1,L_1,p_1.c_1,\mathcal{G}_1)$ and $\mathcal{O}_2(o_2) = (V_2,L_2,p_2.c_2,\mathcal{G}_2)$, then
  - ▹ $V_1 = V_2$ and $L_1 = L_2$
  - ▹ $\mathsf{fields}^L_{K_1X_1}(\mathcal{G}_1) \equiv^{\rho} \mathsf{fields}^L_{K_2X_2}(\mathcal{G}_2)$

▷ If $L_1 = L$ then $p_1.c_1 = p_2.c_2$ else $\mathsf{typeabs}_{K_1X_1}(p_1.c_1) = \mathsf{abs}_{X_1}(\mathsf{typeabs}_{K_2X_2}(p_2.c_2))$

In the following, we explain the definition of $\preccurlyeq_L^\rho$. We first require that there is a renaming from the exposed objects of the first to the exposed objects of the second configuration. Note that exposed objects are exactly those that have so far appeared in the traces. We also require that there is a renaming between the (internal) objects that are created by L. We then require that for both configurations the execution is at the same place (either in code of the library or the context). Furthermore, we require the parts of the stack that consist of code from L to be equivalent under the object renaming. For related objects, the heap entries must also match in the following way. The exposure and location flags must be the same. The values of fields that are defined in L must be equivalent under the object renaming. At last, the dynamic type of related objects must be equal if they are created by L. Otherwise, they must have the same abstracted types. We confirm in the following lemmas that initial states are related under $\preccurlyeq_L$.

Lemma 7.3 (Initial states are related under $\preccurlyeq_{\mathsf{lib}}$)**.** *Consider two program contexts K_1 and K_2 and a library implementation X such that $\vdash K_1X$ and $\vdash K_2X$. Then $\mathcal{S}^{\mathbf{init}}_{K_1X} \preccurlyeq_{\mathsf{lib}} \mathcal{S}^{\mathbf{init}}_{K_2X}$.*

Lemma 7.4 (Initial states are related under $\preccurlyeq_{\mathsf{ctxt}}$)**.** *Consider two library implementations X_1 and X_2 and a program context K such that $\vdash KX_1$ and $\vdash KX_2$ and X_2 is source compatible with X_1. Then $\mathcal{S}^{\mathbf{init}}_{KX_1} \preccurlyeq_{\mathsf{ctxt}} \mathcal{S}^{\mathbf{init}}_{KX_2}$.*

We first present how the relations $\preccurlyeq_L^\rho$ are affected by steps of the small-step operational semantics and later extend it to large steps. Before we can proceed, we need to extend our terminology by the notions of *minimal* renaming and *consistency* between renamings.

Definition 7.3 (Minimality and consistency of renamings)**.** A renaming ρ is minimal for a property P if ρ satisfies P and no other $\rho' \subset \rho$ exists which satisfies P. Two renamings are consistent if the union of both relations yields a renaming again, i.e., they agree on the common value pairs.

Note that consistency is an equivalence relation. Composition ($\circ$) of two renamings yields a renaming again. Union ($\cup$) of two renamings that are consistent with each other yields a renaming again.

7.2.1. Small-step Semantics

To study the effect of the relations $\preccurlyeq_L$ on the small-step operational semantics, we consider three different cases. We distinguish whether the steps are initiated from L or from $\neg L$. For steps initiated from $\neg L$, we also distinguish whether the steps are labelled by τ or another label. For illustration purposes, we depict the situations graphically.

Lemma 7.5 ($\preccurlyeq_L$ simulates small step from L)**.** *If $\mathcal{S}_1 \preccurlyeq_L^\rho \mathcal{S}_2$ and $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}_1'$ and $\mathsf{exec}(\mathcal{S}_1) = L$, then $\mathcal{S}_2 \overset{\gamma_2}{\rightsquigarrow} \mathcal{S}_2'$ and $\gamma_1 \equiv^{\rho_\gamma} \mathsf{abs}_{X_1}(\gamma_2)$ and ρ_γ minimal and consistent with ρ and $\mathcal{S}_1' \preccurlyeq_L^{\rho\cup\rho_\gamma} \mathcal{S}_2'$.*

$$\begin{tikzcd}
\mathsf{exec}(\mathcal{S}_1)=L \arrow[dr, dash] & & & & & \\
& \mathcal{S}_1 \arrow[r, dotted, "\preccurlyeq_L"] \arrow[d, squiggly, "\gamma_1"] & \mathcal{S}_2 & & \mathcal{S}_1 \arrow[r, dotted, "\preccurlyeq_L"] \arrow[d, squiggly, "\gamma_1"] & \mathcal{S}_2 \arrow[d, squiggly, "\gamma_2"] \\
\text{If} & \mathcal{S}_1' & & \text{then} & \mathcal{S}_1' \arrow[r, dotted, "\preccurlyeq_L"] & \mathcal{S}_2'
\end{tikzcd}$$

Proof: By case distinction on reduction rule used. □

Lemma 7.6 (τ steps in $\neg L$ preserve $\preccurlyeq_L$)**.** *Assume that $\mathcal{S}_1 \preccurlyeq_L^\rho \mathcal{S}_2$ and $\mathsf{exec}(\mathcal{S}_1) = \neg L$. If $\mathcal{S}_1 \overset{\tau}{\rightsquigarrow} \mathcal{S}_1'$ then $\mathcal{S}_1' \preccurlyeq_L^\rho \mathcal{S}_2$. Similarly, if $\mathcal{S}_2 \overset{\tau}{\rightsquigarrow} \mathcal{S}_2'$ then $\mathcal{S}_1 \preccurlyeq_L^\rho \mathcal{S}_2'$.*

$\mathsf{exec}(\mathcal{S}_1) = \neg L$

If
$$\begin{tikzcd}
\mathcal{S}_1 \arrow[r, phantom, "\preccurlyeq_L"] \arrow[d, squiggly, "\tau"] & \mathcal{S}_2 \\
\mathcal{S}'_1 &
\end{tikzcd}$$
then
$$\begin{tikzcd}
\mathcal{S}_1 \arrow[r, phantom, "\preccurlyeq_L"] \arrow[d, squiggly, "\tau"] & \mathcal{S}_2 \\
\mathcal{S}'_1 \arrow[ur, phantom, "\preccurlyeq_L"] &
\end{tikzcd}$$

Proof: By case distinction on reduction rule used. □

Lemma 7.7 (Similar labels from $\neg L$ preserve $\preccurlyeq_L$). *If $\mathcal{S}_1 \preccurlyeq^{\rho}_L \mathcal{S}_2$ and $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}'_1$ and* $\mathsf{exec}(\mathcal{S}_1) = \neg L$ *and $\mathcal{S}_2 \overset{\gamma_2}{\rightsquigarrow} \mathcal{S}'_2$ and $\gamma_1 \equiv^{\rho_\gamma}$* $\mathsf{abs}_{X_1}(\gamma_2) \neq \tau$ *and ρ_γ minimal and consistent with ρ, then $\mathcal{S}'_1 \preccurlyeq^{\rho \cup \rho_\gamma}_L \mathcal{S}'_2$.*

$\mathsf{exec}(\mathcal{S}_1) = \neg L$

If
$$\begin{tikzcd}
\mathcal{S}_1 \arrow[r, phantom, "\preccurlyeq_L"] \arrow[d, squiggly, "\gamma_1 \neq \tau"] & \mathcal{S}_2 \arrow[d, squiggly, "\gamma_2"] \\
\mathcal{S}'_1 & \mathcal{S}'_2
\end{tikzcd}$$
with $\gamma_1 \equiv \gamma_2$,
then
$$\begin{tikzcd}
\mathcal{S}_1 \arrow[r, phantom, "\preccurlyeq_L"] \arrow[d, squiggly, "\gamma_1"] & \mathcal{S}_2 \arrow[d, squiggly, "\gamma_2"] \\
\mathcal{S}'_1 \arrow[r, phantom, "\preccurlyeq_L"] & \mathcal{S}'_2
\end{tikzcd}$$
with $\gamma_1 \equiv \gamma_2$

Proof: By case distinction on reduction rule used. □

7.2.2. Large-step Semantics

The three lemmas of the previous subsection can be extended to large steps and then to many large steps, i.e., (partial) program runs.

Lemma 7.8 ($\preccurlyeq_L$ simulates large step from L). *If $\mathcal{S}_1 \preccurlyeq^{\rho}_L \mathcal{S}_2$ and $\mathcal{S}_1 \xrightarrow{\gamma_1} \mathcal{S}'_1$ and* $\mathsf{exec}(\mathcal{S}_1) = L$*, then $\mathcal{S}_2 \xrightarrow{\gamma_2} \mathcal{S}'_2$ and $\gamma_1 \equiv^{\rho_\gamma}$* $\mathsf{abs}_{X_1}(\gamma_2)$ *and ρ_γ minimal and consistent with ρ and $\mathcal{S}'_1 \preccurlyeq^{\rho \cup \rho_\gamma}_L \mathcal{S}'_2$.*

Proof: By induction on the number of small steps and Lemma 7.5. □

Lemma 7.9 (Similar large step from $\neg L$ preserves $\preccurlyeq_L$). *If $\mathcal{S}_1 \preccurlyeq^{\rho}_L \mathcal{S}_2$ and $\mathcal{S}_1 \xrightarrow{\gamma_1} \mathcal{S}'_1$ and* $\mathsf{exec}(\mathcal{S}_1) = \neg L$ *and $\mathcal{S}_2 \xrightarrow{\gamma_2} \mathcal{S}'_2$ and $\gamma_1 \equiv^{\rho_\gamma}$* $\mathsf{abs}_{X_1}(\gamma_2)$ *and ρ_γ minimal and consistent with ρ, then $\mathcal{S}'_1 \preccurlyeq^{\rho \cup \rho_\gamma}_L \mathcal{S}'_2$.*

Proof: By induction on the number of τ steps and Lemma 7.6 and Lemma 7.7. □

We then relate many large steps. For deterministic contexts, we can state that if we have a run starting from a state and another run starting from a related state which emits a trace equivalent to the first one, then the end states are related. We generalize this in the following corollary, where we also consider non-deterministic (i.e. most general) contexts.

Corollary 7.10 ($\preccurlyeq_L$ simulates multiple large steps). *If $\mathcal{S}_1 \preccurlyeq^{\rho}_L \mathcal{S}_2$ and $\mathcal{S}_1 \xrightarrow{\overline{\gamma}_1} \mathcal{S}'_1$ and $\mathcal{S}_2 \xrightarrow{\overline{\gamma}_2} \mathcal{S}'_2$ and $\overline{\gamma}_1 \equiv^{\rho^{12}_{\overline{\gamma}}}$* $\mathsf{abs}_{X_1}(\overline{\gamma}_2)$ *and $\rho^{12}_{\overline{\gamma}}$ minimal and consistent with ρ, then $\exists \mathcal{S}'_3$ such that $\mathcal{S}_2 \xrightarrow{\overline{\gamma}_3} \mathcal{S}'_3$ and $\overline{\gamma}_1 \equiv^{\rho^{13}_{\overline{\gamma}}}$* $\mathsf{abs}_{X_1}(\overline{\gamma}_3)$ *and $\mathcal{S}'_1 \preccurlyeq^{\rho \cup \rho^{13}_{\overline{\gamma}}}_L \mathcal{S}'_3$.*

Proof: Directly from Lemma A.6, which provides a stronger consequent needed for the induction step. The proof then goes by induction on the length of trace $\overline{\gamma}_1$. □

The states $\mathcal{S}'_1$ and $\mathcal{S}'_2$ might not be related, as during the runs $\mathcal{S}_1 \xrightarrow{\gamma_1} \mathcal{S}'_1$ and $\mathcal{S}_2 \xrightarrow{\gamma_2} \mathcal{S}'_2$, the same most general context might have chosen different executions as it is non-deterministic. For example, it may create more objects that are internal to it in one execution than in another, but still generate an equivalent trace. For deterministic contexts, however, $\mathcal{S}'_1 \preccurlyeq_L \mathcal{S}'_2$.

The proofs of Lemmas 6.1 and 6.2 follow directly from the previous lemmas.

7.3. Sound MGC Abstraction

In this section we define the simulation relation to prove Lemma 6.3. The goal is to prove that $\mathsf{traces}(KX) \subseteq \mathsf{traces}(\mathsf{mgc}(X)X)$. We thus give a simulation relation on runtime configurations that relates the configurations of KX and $\mathsf{mgc}(X)X$ such that whenever for related states the first configuration can make a step, the second one can make the same step (plus a few τ steps). This relation is defined as $\preccurlyeq_{\mathsf{lib}} \cap \lll$.

We first give a function that abstracts interaction frames of deterministic program contexts to their MGC counterpart.

Definition 7.4 (MGC stack abstraction $\mathsf{nondet}(\overline{\mathcal{F}})$).

$$\mathsf{nondet}(\overline{\mathcal{F}}) \overset{\text{def}}{\equiv} \begin{cases} \bullet & \text{if } \overline{\mathcal{F}} = \bullet \\ \mathsf{nde}_{\mathsf{ctxt}}{:}p.t \cdot \mathsf{nondet}(\overline{\mathcal{F}}') & \text{if } \overline{\mathcal{F}} = E_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}}' \\ \mathsf{let\ lang.Object}\ x = [\,]\ \mathsf{in}\ \mathsf{nde}_{\mathsf{ctxt}}{:}p.t \cdot \mathsf{nondet}(\overline{\mathcal{F}}') & \text{if } \overline{\mathcal{F}} = \mathcal{E}_{\mathsf{ctxt}}{:}p.t \cdot \overline{\mathcal{F}}' \\ \overline{\mathcal{F}} & \text{if } \overline{\mathcal{F}} = \mathsf{null}_{\mathsf{ctxt}}{:}\bot \end{cases}$$

Note that the choice of the identifier x can be normalized. We omit the details for simplicity. The last case considers the terminal configuration after **R-Success**. We define the relation $\lll$, which relates the deterministic program context to the MGC, as follows:

Definition 7.5 (Preorder relation $\lll$). Consider two well-formed configurations $\mathcal{S}_1 = KX, \mathcal{O}_1, \overline{\mathcal{F}_1}$ and $\mathcal{S}_2 = \mathsf{mgc}(X)X, \mathcal{O}_2, \overline{\mathcal{F}_2}$. We write $\mathcal{S}_1 \lll^{\rho_e} \mathcal{S}_2$ if ρ_e is a bijective renaming from $\mathsf{filter}(\mathcal{O}_1, \mathsf{exposed})$ to $\mathsf{filter}(\mathcal{O}_2, \mathsf{exposed})$ and ρ_i is a bijective renaming from $\mathsf{filter}(\mathcal{O}_1, \mathsf{internal}, \mathsf{ctxt})$ to $\mathsf{filter}(\mathcal{O}_2, \mathsf{internal}, \mathsf{ctxt})$ and $\rho = \rho_e \cup \rho_i$ such that
- ▹ $\mathsf{nondet}(\mathsf{stackabs}_{\mathsf{ctxt}}(\overline{\mathcal{F}_1})) \equiv^{\rho} \mathsf{stackabs}_{\mathsf{ctxt}}(\overline{\mathcal{F}_2})$
- ▹ If $o_1 \equiv^{\rho} o_2$ and $\mathcal{O}_1(o_1) = (V_1, L_1, p_1.c_1, \mathcal{G}_1)$ and $\mathcal{O}_2(o_2) = (V_2, L_2, p_2.c_2, \mathcal{G}_2)$, then:
  - ▹ $V_1 = V_2$ and $L_1 = L_2$
  - ▹ $\mathsf{typeabs}_{KX}(p_1.c_1) = \mathsf{typeabs}_{\mathsf{mgc}(X)X}(p_2.c_2)$

The main requirements of the relation $\lll$ are that the stack of the MGC corresponds to the abstracted stack of the deterministic program context and that related objects have the same abstracted type. We confirm that the relation $\preccurlyeq_{\mathsf{lib}} \cap \lll$ has the simulation property.

Lemma 7.11 (Initial states are related under $\preccurlyeq_{\mathsf{lib}} \cap \lll$). *Consider a deterministic program context K and a library implementation X such that $\vdash KX$. Then $\mathcal{S}^{\mathbf{init}}_{KX} \preccurlyeq_{\mathsf{lib}} \cap \lll \mathcal{S}^{\mathbf{init}}_{\mathsf{mgc}(X)X}$.*

Proof: Trivial. □

Note that sometimes two operational steps are needed by the MGC to simulate one operational step of the deterministic program context. Steps by the deterministic context using the rule **R-New** are simulated by the MGC using the rule **MGC-New**. Steps by the context using the rule **R-Call-Boundary** are simulated by applying the rules **MGC-Prepare-Call** and **R-Call-Boundary**. Steps by the context using the rule **R-Return-Boundary** are simulated using the rules **MGC-Prepare-Return** and **R-Return-Boundary**. Finally, all other steps by the context are simulated using the rule **MGC-Skip**. Steps by the library using any rule are simulated using the same rule (except **R-Return-Boundary** which is simulated by **R-Return-Boundary** and **R-Let**, as interaction frames of the MGC are of the form $\mathsf{let\ lang.Object}\ x = v\ \mathsf{in\ nde}$ due to **MGC-Prepare-Call**).

Lemma 7.12 ($\preccurlyeq_{\mathsf{lib}} \cap \lll$ simulates small steps). *If $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{lib}} \cap \lll^{\rho} \mathcal{S}_2$ and $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}_1'$, then ($\mathcal{S}_2 \overset{\gamma_2}{\rightsquigarrow} \mathcal{S}_2'$ or $\mathcal{S}_2 \overset{\tau}{\rightsquigarrow} \_ \overset{\gamma_2}{\rightsquigarrow} \mathcal{S}_2'$ or $\mathcal{S}_2 \overset{\gamma_2}{\rightsquigarrow} \_ \overset{\tau}{\rightsquigarrow} \mathcal{S}_2'$) and $\gamma_1 \equiv^{\rho_\gamma} \gamma_2$ and ρ_γ minimal and consistent with ρ and $\mathcal{S}_1' \preccurlyeq^{\rho \cup \rho_\gamma}_{\mathsf{lib}} \cap \lll^{\rho \cup \rho_\gamma} \mathcal{S}_2'$.*

Proof: By case distinction on reduction rule used. Lemmas 7.5 to 7.7 are also used. □

7.4. Complete MGC Abstraction

We give the construction of a deterministic program context in Section 7.4.1 to simulate a run of the most general context and define the simulation relation $\ggg$ in Section 7.4.2 to prove Lemma 6.4. The story goes as follows: Let X be a library implementation and $\overline{\gamma} \in \mathsf{traces}(\mathsf{mgc}(X)X)$. We consider a run of $\mathsf{mgc}(X)X$ which simulates this $\overline{\gamma}$ and show that we can construct a deterministic program context K which simulates this run and leads to the same trace. The simulation relation is defined as $\preccurlyeq_{\mathsf{lib}} \cap \ggg$.

```
package main;
public class Main {
  main.Main $f_{\mathsf{main}}$;
  $p_0.c_0\ f_0^{\mathsf{ctxt}}; \ldots\ p_{nx}.c_{nx}\ f_{nx}^{\mathsf{ctxt}}$;
  lang.Object $f_0^{\mathsf{lib}}$; ... lang.Object $f_{nm}^{\mathsf{lib}}$;
  main.Number $f_0^N$; ... main.Number $f_z^N$;
  main.Number $f_{\mathsf{currNum}}$;
  getter−methods for all $f_j^N$ and $f_{\mathsf{currNum}}$
  getter− and setter−methods for all $f_j^{\mathsf{ctxt}}$ and $f_j^{\mathsf{lib}}$
  lang.Object incrNum() { ... }
  lang.Object main() {
    this.$f_{\mathsf{main}}$ = this;
    initialize $f_j^N$ fields with distinct Number objects
    this.$f_{\mathsf{currNum}}$ = this.$f_0^N$;
    NODE$_0$
  }
}
public class Number {}
```

Figure 15: main package of constructed context

7.4.1. Construction of Deterministic Program Context

We construct a deterministic program context based on a (partial) run of the most general context. Running the constructed program context should lead to an equivalent trace. The high level idea behind the construction is the following. The class structure for the constructed context K is nearly the same as for mgc(X) (except method bodies and a few extra fields and methods). The method bodies however now contain expressions that simulate the choices made by the most general context (when executing the non-deterministic expression nde). As the choices may differ for different method incarnations, the construction needs to account for this and distinguish the different incarnations. This is done by having a bookkeeping object (which every object of a type of the context refers to) that globally counts method incarnations of methods defined in the context. To enable access to all visible objects (**MGC-Prepare-Call** and **MGC-Prepare-Return**), the bookkeeping object has extra fields to store references to all objects that have been created so far by the program context or that have been exposed by the library.

We start by constructing the class structure of K which is similar as the one for the construction of the most general context in Section 5.3. For each class $p_{\mathbf{mgc}}.c$ in mgc(X) except main.Main, we construct a class with same name and header of the form

public class c extds $p_0.c_0$ impls $\overline{p.i}$ {main.Main f_{main}; lang.Object setMain(main.Main x){ this.$f_{\mathsf{main}} = x$ } $\widehat{M^{\mathsf{c}}}$}

where (1) f_{main} is a field name to refer to the initial object of class main.Main, (2) setMain is a method name not occurring anywhere in X, (3) $\widehat{M^{\mathsf{c}}}$ is the set of methods corresponding to the ones of the class $p_{\mathbf{mgc}}.c$ in mgc(X), but with different method bodies. The method bodies are explained after the description of the class main.Main.

The class main.Main (see Figure 15) is the startup class and plays the bookkeeping role. It has field declarations $p_j.c_j\ f_j^{\mathsf{ctxt}}$; for every object that is created in the (partial) run by the most general context, where $p_j.c_j$ is the dynamic class type of the object. For every object of the library that is exposed in the run, we declare a field f_j^{lib} of type lang.Object. We cannot provide a more specific type to these fields, because the dynamic type of these objects might be local to a package of X and thus not accessible in K. When we use the objects for method calls or returns, we cast to an appropriate type, however. We also provide getter and setter methods for the f_j^{ctxt} and f_j^{lib} fields.

To count method incarnations, we use objects of an additional Number class which we define in the package main. Each number is represented by a different Number object. We add a field main.Number f_j^N; for each number that refers to the corresponding Number object. The field f_{currNum} stores a reference to the object representing the number of the current method incarnation. The method incrNum sets the f_{currNum} field to the object representing the next number (it is implemented as a huge if-else cascade). We also provide getter methods for the f_j^N fields and f_{currNum}. In the main method, we initialize the Number fields.

Using the given definition of the main.Main class, we can now implement the method bodies of $\widehat{M^{\mathsf{c}}}$. They have the following form, where we use if-else syntax for better readability:

```
this.$f_{\mathsf{main}}$.incrNum();
```

```
if (this.$f_{\text{main}}$.get$f_{\text{currNum}}$() == this.$f_{\text{main}}$.get$f^{N}_{j_1}$()) { NODE$_{j_1}$ }
else if ... { ... }
else if (this.$f_{\text{main}}$.get$f_{\text{currNum}}$() == this.$f_{\text{main}}$.get$f^{N}_{j_n}$()) { NODE$_{j_n}$ }
else { null } // never reached
```

For each invocation of a method in the context, we increase the method incarnation counter. Then we choose the right code NODE_j to execute based on which method incarnation we currently are simulating. Each NODE_j will only be executed once and its construction is based on the steps done by the MGC when reducing the expression nde in the corresponding method incarnation. For each step, an expression is created, which thus leads to a sequence of expressions (delimited by ;). In the following we explain the construction of NODE_j.

First, NODE_j starts by storing the transmitted values (parameters of the method call for this method incarnation j) if necessary, i.e., if these values have not been stored already by an earlier NODE. We call these actions $\mathsf{STORE}(x)$ and they have the following form: $\mathsf{STORE}(x)$ is null if we do not care about the value pointed to by x, i.e. if this value was already stored, or if the value is null. We use the expression this.f_{main}.set$f_j^{\text{lib}}(x)$ for library objects that have not been previously stored and the expression this.f_{main}.set $f_j^{\text{ctxt}}((p.c)x$ err null) for context objects of dynamic type $p.c$ that have not been previously stored. Note that j is the index to the right field we use to store the object. Note that for each object that occurs in visible($\mathcal{O}$, ctxt) in the program run, there is exactly one STORE instruction in the constructed program context K. Also note that we need to put STORE(this) at the beginning of NODE_0.

After the storage, we now construct for each step by the MGC an expression which simulates the step. We thus get a sequence of expressions in NODE_j. We start with the simple cases:

- ▹ For a step with **MGC-New**, we use the expression let $p.c$ x = new $p.c$ in (x.setMain(this.f_{main}); STORE(x)) where x is a fresh variable name not appearing elsewhere in the construction.
- ▹ For **MGC-Skip**, we use the expression (main.Main)null err null (arbitrarily chosen expression which does not have any observable effect).
- ▹ For **MGC-Prepare-Success**, we use the expression success.

For the cases **MGC-Prepare-Call** and **MGC-Prepare-Return**, we need to access appropriate values in visible($\mathcal{O}$, ctxt). This is done by accessing the corresponding field on the bookkeeping object, which we denote by $v_{\text{corr}} \overset{\text{def}}{=}$ this. f_{main}.getf_j^{ctxt}(), this.f_{main}.getf_j^{lib}() or null.

- ▹ For **MGC-Prepare-Return**, we use $(p.t)v_{\text{corr}}$ err null where $p.t$ is the return type of the enclosing method.
- ▹ For **MGC-Prepare-Call**, we use let lang.Object $x = ((p_0.t_0)v_{\text{corr}}$ err null).$m(\overline{(p.t)v_{\text{corr}}\text{ err null}})$ in STORE(x) where $p_0.t_0$ is one of the types of typeabs$_{KX}(p.c)$ such that $\langle m, p_2.t_2 \cdot \overline{p.t}, \_\rangle \in_X p_0.t_0$, and x is a fresh variable name not appearing elsewhere.

7.4.2. Simulation

Using the previous construction, we can then define the simulation relation between the most general and the constructed program context.

Definition 7.6 (Preorder relation $\ggg$). Consider two well-formed configurations $\mathcal{S}_1 = \text{mgc}(X)X, \mathcal{O}_1, \overline{\mathcal{F}_1}$ and $\mathcal{S}_2 = KX, \mathcal{O}_2, \overline{\mathcal{F}_2}$ where K is constructed using the construction described in Section 7.4.1. We write $\mathcal{S}_1 \ggg^{\rho_e} \mathcal{S}_2$ if ρ_e is a bijective renaming from filter($\mathcal{O}_1$, exposed) to filter($\mathcal{O}_2$, exposed) and ρ_i is a bijective renaming from filter($\mathcal{O}_1$, internal, ctxt) to the subset of filter($\mathcal{O}_2$, internal, ctxt) with class type different to main.Number and $\rho = \rho_e \cup \rho_i$ such that

- ▹ stackabs$_{\text{ctxt}}(\overline{\mathcal{F}_1}) \equiv^{\rho}$ nondet(stackabs$_{\text{ctxt}}(\overline{\mathcal{F}_2})$)
- ▹ If $o_1 \equiv^{\rho} o_2$ and $\mathcal{O}_1(o_1) = (V_1, L_1, p_1.c_1, \mathcal{G}_1)$ and $\mathcal{O}_2(o_2) = (V_2, L_2, p_2.c_2, \mathcal{G}_2)$:
  - ▹ $V_1 = V_2$ and $L_1 = L_2$
  - ▹ $p_1.c_1 = p_2.c_2$
- ▹ $\exists o_{\text{main}}$ such that $\mathcal{O}_2(o_{\text{main}}) =$ (internal, ctxt, main.Main, $\mathcal{G}$) and
  - ▹ objectrefs($\mathcal{G}(\overline{\text{main.Main}, f^{\text{ctxt}}})$) $\equiv^{\rho}$ filter($\mathcal{O}_1$, ctxt)
  - ▹ objectrefs($\mathcal{G}(\overline{\text{main.Main}, f^{\text{lib}}})$) $\equiv^{\rho}$ filter($\mathcal{O}_1$, exposed, lib)

- ▹ $\mathcal{G}(\overline{f^N})$ holds the main.Number objects, which are distinct each. $\mathcal{G}(f_{\mathsf{currNum}})$ represents the current node and holds an object from $\mathcal{G}(\overline{f^N})$.
- ▹ $\forall o \in \mathsf{dom}(\mathcal{O}_2)$ with $\mathcal{O}_2(o) = (\_, \mathsf{ctxt}, p.c, \mathcal{G}_2)$ and $p.c \in \mathcal{C}_K$ and $p.c \neq$ main.Number, we have $\mathcal{G}_2(p.c, f_{\mathsf{main}}) = o_{\mathsf{main}}$.

On one hand, we have a coupling that is stronger than $\lll$, as the constructed context uses exactly the same classes as the abstract context, just with other method bodies. On the other hand, the coupling is weaker as it does not relate all internal objects of ctxt (i.e. the helper objects representing numbering). To show the simulation property of $\preccurlyeq_{\mathsf{lib}} \cap \ggg$, similar as for Section 7.3, we first show that initial states are related and that the relation is preserved by small steps. We assume that the steps that are done by the MGC are those upon which we based the construction of the deterministic program context.

The constructed program context needs to initialize first before it can simulate the MGC.

Lemma 7.13 (Initial states are related under $\preccurlyeq_{\mathsf{lib}} \cap \ggg$ after a few steps). *Consider a library implementation X and a deterministic program context K constructed from a run of* $\mathsf{mgc}(X)X$. *Then* $\mathcal{S}^{\mathbf{init}}_{KX} \overbrace{\overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow}}^{i \text{ times, } i \in \mathbb{N}} \mathcal{S}$ *and* $\mathcal{S}^{\mathbf{init}}_{\mathsf{mgc}(X)X} \preccurlyeq_{\mathsf{lib}} \cap \ggg \mathcal{S}$.

Often, many steps are needed by the constructed context to simulate a step of the MGC. A technical challenge is that there are steps by the MGC for which the relation $\lll$ can not be established, e.g. **MGC-Prepare-Call** yields an interaction frame that is not in the range of nondet(). In that case, we know that there is always a unique next step for which the relation can be established. Steps that are guaranteed to occur after each other are **MGC-Prepare-Call** and **R-Call-Boundary**, **MGC-Prepare-Success** and **R-Success**, or **R-Return-Boundary** and **R-Let** if it is a return to the MGC. In the following simulation lemma we distinguish three disjoint cases (which are exhaustive).

Lemma 7.14 ($\preccurlyeq_{\mathsf{lib}} \cap \ggg$ simulates small steps). *Consider $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{lib}} \cap \ggg^{\rho} \mathcal{S}_2$. If $\mathcal{S}_1 \overset{\gamma}{\rightsquigarrow} \mathcal{S}'_1$ is a step using the rule* **MGC-Prepare-Call** *or* **MGC-Prepare-Success**, *then $\gamma = \tau$ and there are unique $\mathcal{S}''_1$, γ_1 such that $\mathcal{S}'_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}''_1$ and $\Big($ $\mathcal{S}_2 \overbrace{\overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow}}^{i \text{ times, } i \in \mathbb{N}} \_ \overset{\gamma_2}{\rightsquigarrow} \_ \overbrace{\overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow} \ldots \overset{\tau}{\rightsquigarrow}}^{j \text{ times, } j \in \mathbb{N}} \mathcal{S}'_2$ and $\gamma_1 \equiv^{\rho_\gamma} \gamma_2$ and ρ_γ minimal and consistent with ρ and $\mathcal{S}''_1 \preccurlyeq^{\rho \cup \rho_\gamma}_{\mathsf{lib}} \cap \ggg^{\rho \cup \rho_\gamma} \mathcal{S}'_2$ $\Big)$ ($\star$). If $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}'_1$ where γ_1 is an return output label (i.e.* rtrn _⇧*), then there are unique $\mathcal{S}''_1$, γ such that $\mathcal{S}'_1 \overset{\gamma}{\rightsquigarrow} \mathcal{S}''_1$ and $\gamma = \tau$ and $\star$. If $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}''_1$ and none of the previous cases apply, then $\star$.*

Proof: By case distinction on reduction rule used. Lemmas 7.5 to 7.7 are also used. □

7.4.3. Differentiating Context

In order to distinguish two library implementations that generate different labels at some point in the trace, we construct a deterministic program context that makes the first configuration succeed and prevents the second one from doing so. To construct such a context, we have to generate program code that can distinguish the situations.

We use the same construction as in Section 7.4.1. However, we also add a method lang.Object loop() { this. loop(); } to the main.Main class. To prevent a program to succeed, we then just call this diverging method by this.f_{main}.loop() in a node where we distinguish the situations.

As our constructed context is deterministic (up to object naming), the constructed program context reaches the same nodes (NODE_j) for equivalent prefixes of the trace. The construction then depends on the last label in which the two traces differ. If different nodes are reached (which is e.g. the case if one label is a call and the other is a return or if they are both calls but with different method names) we put in the first node success and in the second one we call this.f_{main}.loop(). If the same node is reached, we have to compare the abstract values that occur at the same position in the labels. Wlog we assume that they differ at a certain place and that this value is referred to by variable x: if one value is null and the other one not, we use the expression ($x ==$ null ? success : this.f_{main}.loop()). We assume in the following that both are abstracted objects and consider the two remaining cases:

- ▹ At least one object has occurred earlier in the trace. Then we compare the objects to the corresponding object stored in the f^{ctxt} or f^{lib} field: $(x == \mathsf{this}.f_{\mathsf{main}}.\mathsf{get}f_j^{\mathsf{lib}}()\ ?\ \mathsf{success} : \mathsf{this}.f_{\mathsf{main}}.\mathsf{loop}())$.
- ▹ Both objects are created by the library and have not occurred earlier in the trace: Then the abstracted types ${T^{\alpha}}_1$ of v_1^{α} and ${T^{\alpha}}_2$ of v_2^{α} have to be different.[4] Wlog let $p.t$ be a public type in ${T^{\alpha}}_1$, but not in ${T^{\alpha}}_2$. Thus, we can distinguish the values by casting to this type: $(((p.t)x\ \mathsf{err\ null}) == \mathsf{null}\ ?\ \mathsf{success} : \mathsf{this}.f_{\mathsf{main}}.\mathsf{loop}())$.

8. Reasoning About Backwards Compatibility

Building on the formal foundations of the previous sections, we outline in the following a method for reasoning about backward compatibility of libraries. As a prerequisite for reasoning in terms of code, we first discuss the relation between traces and program code in Section 8.1. The reasoning method, presented in Section 8.2, is then based on the idea of directly connecting the representations of both library implementations using a so-called *coupling relation* [36–38].

8.1. From Traces to Program Code

An important part of the reasoning method is to establish a relation between the traces and the program code. If we have a call input label, we need to find out what the possible targets of such a call are, i.e., the method bodies in the library implementation resulting from a method dispatch that leads to such a label. Similarly, if we have a return input label, we need to find the places in the library implementation where we can return to. In general, these places cannot statically be determined, as we illustrate in the following example. Consider a public class C and a local class D as part of a library implementation where the method m in D overrides the method m in C.

```
public class C { public void m() { BODY1 } }
class D extends C { public void m() { BODY2 } }
```

Also consider a program context from which the class D is not accessible (i.e. defined in another package). If a D object is exposed under the C type, then a change in control can reach BODY2. This is the case, e.g., if there is a method of the following form in the library implementation:

```
public class Factory { public static C instance() { return new D(); } }
```

If an input label of the form $\mathsf{call}\ o^{\alpha}.m()⊞$ occurs where the abstracted type of o^{α} is $\langle\{\mathsf{C}\},\bullet\rangle$, then the program could, depending on the actual runtime type, either lead to a dispatch of the method m defined in C or in D (i.e. BODY1 or BODY2). The places which are targets of a change in control can be approximated based on the types that appear in the input/output labels. As the labels are directly based on the runtime type information, we can give an inverse of the abstraction function $\mathsf{typeabs}_{KX}(p.c)$ that computes the abstracted types of the labels. This inverse is a relation, i.e., it associates multiple places in the library implementation with a certain shape of message.

A more precise static analysis in the likes of [39] can be used to statically determine the shape of messages in most cases. Remaining (open) cases have to be formulated as part of a program invariant. For example, the invariant may specify the property that there are no exposed objects of dynamic type D. In that case, input labels could never contain D objects and thus never dispatch to a method in D.

8.2. Simulation-based Proof Method

Before describing the reasoning method, we recapitulate the proof obligations that are needed in order to prove two library implementations compatible. We also describe how the direct connection of the trace semantics to the operational semantics can be exploited to prove compatibility.

[4]Note that for objects created by the library, their dynamic class type is always defined in the library implementation (Definition 7.1), and thus the set $\widehat{m}$ of the abstracted type $T^{\alpha} \overset{\text{def}}{=} \langle\widehat{p.t},\widehat{m}\rangle$ of the object is uniquely determined by $\widehat{p.t}$.

In order to prove that a library implementation Y is backward compatible with a library implementation X using the trace-based definition of compatibility, the following steps are necessary. First, Y must be proven source compatible with X. This can be directly done by the checks detailed in Section 4. The more difficult part is to prove, as per Definition 6.4, that $\mathsf{traces}(\mathsf{mgc}(X)X) \subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$. By Lemma 6.5, we know that it is sufficient to prove that $\mathsf{traces}(\mathsf{mgc}(X)X) \subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$.

In the following, we present an approach for proving backward compatibility based on the specialized simulation relations introduced earlier. Similar as for the full abstraction proof, we use specialized simulations for reasoning about backward compatibility. From Lemmas 7.4 and 7.5 and Corollary 7.10 we get the property that there is a relation $\preccurlyeq^{\rho}_{\mathsf{ctxt}}$ between ctxt parts of corresponding states of the two library implementations whenever these implementations are trace compatible. The relation ensures that large steps from the context are simulated properly (Lemma 7.8). In order to prove trace inclusion, we then need to prove that large steps from related states in the library implementations are also simulated properly. For this, we need to relate also the parts of the configurations which belong to the respective library implementations. Such a relation is called a *coupling relation*. The relation can rely on the following properties of $\preccurlyeq^{\rho}_{\mathsf{ctxt}}$ (see Definition 7.2). There is a renaming ρ from the exposed objects of the first configuration to the exposed objects of the second (i.e. the objects occurring in the trace). We call this renaming a *correspondence relation*. This relation between objects of the different runtime configurations can be exploited to relate two implementations of a library, namely, we can talk about corresponding objects, which are those, that appear at the same positions in both traces.

To prove trace compatibility, a coupling relation needs to be provided. The coupling relation between the states of both library implementations can be described with the help of the correspondence relation ρ. We then need to prove for coupled states that the next labels of small steps in the context or large steps in the library are also related and the states coupled and that the coupling holds for the initial states of the programs. We introduce the notion of *adequate* coupling relation to denote coupling relations that have this simulation property.

Definition 8.1 (Adequate coupling). $\preccurlyeq_{\mathbf{inv}}$ is an *adequate* coupling relation for two source compatible library implementations X and Y if

- ▷ $\mathcal{S}^{\mathbf{init}}_{\mathsf{mgc}(X)X} \preccurlyeq_{\mathsf{ctxt}} \cap \preccurlyeq_{\mathbf{inv}} \mathcal{S}^{\mathbf{init}}_{\mathsf{mgc}(X)Y}$, and
- ▷ If $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho}_{\mathbf{inv}} \mathcal{S}_2$ and $\mathsf{exec}(\mathcal{S}_1) = \mathsf{ctxt}$ and $\mathcal{S}_1 \overset{\gamma_1}{\rightsquigarrow} \mathcal{S}_1'$, then $\mathcal{S}_2 \overset{\gamma_2}{\rightsquigarrow} \mathcal{S}_2'$ and $\gamma_1 \equiv^{\rho_\gamma} \mathsf{abs}_X(\gamma_2)$ and ρ_γ minimal and consistent with ρ and $\mathcal{S}_1' \preccurlyeq^{\rho\cup\rho_\gamma}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho\cup\rho_\gamma}_{\mathbf{inv}} \mathcal{S}_2'$, and
- ▷ If $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho}_{\mathbf{inv}} \mathcal{S}_2$ and $\mathsf{exec}(\mathcal{S}_1) = \mathsf{lib}$ and $\mathcal{S}_1 \xrightarrow{\gamma_1} \mathcal{S}_1'$, then $\mathcal{S}_2 \xrightarrow{\gamma_2} \mathcal{S}_2'$ and $\gamma_1 \equiv^{\rho_\gamma} \mathsf{abs}_X(\gamma_2)$ and ρ_γ minimal and consistent with ρ and $\mathcal{S}_1' \preccurlyeq^{\rho\cup\rho_\gamma}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho\cup\rho_\gamma}_{\mathbf{inv}} \mathcal{S}_2'$.

In general, a coupling relation only needs to talk about the library part of the configuration. If the relation only relates the lib parts, then we also have the guarantee that it is preserved by the context, which allows us to disregard steps in the (most general) context in the proof.

We can show that a coupling relation always exists if two library implementations are backward compatible. This completeness result comes basically for free from our full abstraction proof using specialized simulations.

Theorem 3 (Soundness and completeness of adequate couplings). *Consider two library implementations X and Y. Then Y is trace compatible with X iff there exists an adequate coupling relation for X and Y.*

PROOF: Soundness follows directly from Definitions 5.4 and 6.4. Completeness follows by construction from Lemmas 7.4 and 7.5 and Corollary 7.10.

In the following, we illustrate coupling relations using the Cell example in Figure 1. The definition of a coupling relation is also called a *coupling invariant*. An intuitive description of the coupling invariant was already given in Section 1. Using the correspondence relation ρ, we can formally talk about corresponding objects in both program runs. For all corresponding objects $(o_1, o_2) \in \rho$ that have the dynamic type Cell or a subtype thereof and where the value of the field o_2.f is true, the values that are stored in the fields o_1.c and o_2.c1 are either both null or corresponding objects, i.e. $(o_1.\mathrm{c}, o_2.\mathrm{c1}) \in \rho$. Similarly, if the value of the field o_2.f is false, then $(o_1.\mathrm{c}, o_2.\mathrm{c2}) \in \rho$ or these fields are both null. We can formalize this as follows. We write $\mathcal{O}(o, p.c, f) \overset{\text{def}}{=} \mathcal{G}(p.c, f)$ if $\mathcal{O}(o) = (\_, \_, \_, \mathcal{G})$ and then define $\preccurlyeq^{\rho}_{\mathbf{inv}}$ as $\{(\mathcal{S}_1, \mathcal{S}_2) \mid \forall (o_1, o_2) \in \rho : \mathsf{type}_{\mathcal{O}_1}(o_1) \leq_{\mathsf{mgc}(X_1)X_1} \mathrm{Cell} \rightarrow$ if $\mathcal{O}_1(o_1, \mathrm{Cell}, \mathrm{f})$ then $\mathcal{O}_1(o_1, \mathrm{Cell}, \mathrm{c}) \equiv^{\rho} \mathcal{O}_2(o_2, \mathrm{Cell}, \mathrm{c1})$ else $\mathcal{O}_1(o_1, \mathrm{Cell}, \mathrm{c}) \equiv^{\rho} \mathcal{O}_2(o_2, \mathrm{Cell}, \mathrm{c2})\}$.

We then show that $\preccurlyeq^{\rho}_{\mathbf{inv}}$ is an adequate coupling relation. Note that, as explained in Section 8.1, a superset of the shapes (i.e., types and method names) of all possible input labels can be derived from the code of the library implementation, e.g., the public methods. For this particular example, the input labels are of the form $\mathsf{call}\ o^{\alpha}.\mathrm{get}()⍗$ and $\mathsf{call}\ o^{\alpha}.\mathrm{set}(v^{\alpha})⍗$. To show that the simulation property is preserved by small steps from the context is done by case distinction over reduction rule used (the reduction rules which are prefixed by MGC). As the coupling invariant talks only about the lib part of the configurations, this proof is trivial. The only interesting case is where related objects which have been created by the most general context with the type Cell or a subtype thereof are exposed to the library. In this case, we know that the fields of both objects are null (see Definition 7.1) and thus the coupling is preserved.

The main proof obligation is to show that the simulation property is preserved by large steps from the library. We consider the states after related inputs (e.g. $\mathsf{call}\ o_1^{\alpha}.\mathrm{get}()⍗$ and $\mathsf{call}\ o_2^{\alpha}.\mathrm{get}()⍗$ if $(o_1, o_2) \in \rho$) in states which are coupled (i.e., where the coupling invariant holds). We then have to prove that the states right after the next change in control are also coupled and the generated (output) labels are related. In the following, we denote the first version of the Cell library by X and the second one by Y. We suffix elements of the configurations of X by 1 and of Y by 2. We only consider the states right after labels of the form $\mathsf{call}\ o^{\alpha}.\mathrm{get}()⍗$, i.e., we have configurations of the form

- ▹ $\mathcal{S}_1 \overset{\text{def}}{=} \mathsf{mgc}(X)X, \mathcal{O}_1, \mathsf{body}_{\mathsf{mgc}(X)X}(\mathrm{Cell}, \mathrm{get})[o_1/\mathsf{this}]_{\mathsf{lib}}{:}\mathsf{lang.Object} \cdot \overline{\mathcal{F}}_1$ and
- ▹ $\mathcal{S}_2 \overset{\text{def}}{=} \mathsf{mgc}(X)Y, \mathcal{O}_2, \mathsf{body}_{\mathsf{mgc}(X)Y}(\mathrm{Cell}, \mathrm{get})[o_2/\mathsf{this}]_{\mathsf{lib}}{:}\mathsf{lang.Object} \cdot \overline{\mathcal{F}}_2$

such that $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho}_{\mathbf{inv}} \mathcal{S}_2$. After large steps, we then have

- ▹ $\mathcal{S}_1 \xrightarrow{\mathsf{rtrn}\ v_1^{\alpha}⍐} \mathsf{mgc}(X)X, \mathcal{O}_1, \overline{\mathcal{F}}'_1$ where $v_1^{\alpha} = \mathsf{valabs}_{\mathsf{mgc}(X)X}(\mathcal{O}_1(o_1, \mathrm{Cell}, \mathrm{c}), \mathcal{O}_1)$, and
- ▹ $\mathcal{S}_2 \xrightarrow{\mathsf{rtrn}\ v_2^{\alpha}⍐} \mathsf{mgc}(X)Y, \mathcal{O}_2, \overline{\mathcal{F}}'_2$ where $v_2^{\alpha} = \begin{cases} \mathsf{valabs}_{\mathsf{mgc}(X)Y}(\mathcal{O}_2(o_2, \mathrm{Cell}, \mathrm{c1}), \mathcal{O}_2) & \text{if } \mathcal{O}_2(o_2, \mathrm{Cell}, \mathrm{f}) \\ \mathsf{valabs}_{\mathsf{mgc}(X)Y}(\mathcal{O}_2(o_2, \mathrm{Cell}, \mathrm{c2}), \mathcal{O}_2) & \text{otherwise} \end{cases}$

From $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathbf{inv}} \mathcal{S}_2$, we get the property then that $\mathsf{rtrn}\ v_1^{\alpha}⍐ \equiv^{\rho} \mathsf{abs}_X(\mathsf{rtrn}\ v_2^{\alpha}⍐)$. As we also had $\mathcal{S}_1 \preccurlyeq^{\rho}_{\mathsf{ctxt}} \mathcal{S}_2$ we get that the successor states are related as well by $\preccurlyeq^{\rho}_{\mathsf{ctxt}} \cap \preccurlyeq^{\rho}_{\mathbf{inv}}$.

9. Conclusion and Future Work

We have presented a fully abstract trace-based semantics for packages of an object-oriented class-based language. We have used specialized simulation relations on an enhanced operational semantics both to prove the full abstraction result and to use as a reasoning method for backward compatibility. We see this as foundational work for relating trace-based specifications to libraries (i.e. sets of classes), verifying library implementations backward compatible or equivalent and proving refactoring transformations behavior preserving. Our approach also illustrates the direct correspondence between the trace-based [10] and simulation-based approaches [19].

We are currently making first steps towards a computer-supported verification technique. Major goals include providing a specification language for describing coupling relations, extending existing program logics for OO programs [40, 41] to our setting and generating proof obligations to be used by interactive or automatic theorem provers. As another direction for future work, we would like to explore an automated testing technique for backward compatibility. Due to our explicit representation of the most general context, we believe this is feasible using a program context that simulates the MGC as a test driver. On a more methodological level, we would like to extend our approach to different settings with varying notions of library or component. In particular, we are interested in studying backward compatibility for libraries in a language with a concurrency model based on concurrently running groups of objects [42, 43]. Another aspect we would like to explore is to study backward compatibility with respect to a restricted set of contexts, as it is not always desirable to preserve the full behavior of libraries in an evolution step. For example fixing bugs or changing parts of the functionality of a library might require us to check that the new version of a library has the same behavior as the old one only with respect to a subset of its interface methods. Weaker compatibilities can be considered for example by (1) providing a more restrictive definition of the most general context (e.g. disallow the context to call a certain method), (2) giving an abstraction function on the traces, or (3) specifying method contracts that must be satisfied by contexts.

Acknowledgements. We thank the anonymous reviewers of SBMF 2011 for their valuable feedback.

[1] D. Dig, R. Johnson, How do APIs evolve? A story of refactoring, Journal of Software Maintenance and Evolution (2006) 83–107.
[2] J. des Rivières, Evolving Java-based APIs, `http://wiki.eclipse.org/Evolving_Java-based_APIs`.
[3] Eclipse PDE API Tools, `http://www.eclipse.org/pde/pde-api-tools/`.
[4] B. Godlin, O. Strichman, Regression verification, in: DAC, ACM, 2009, pp. 466–471.
[5] R. Milner, Fully abstract models of typed lambda-calculi, Theoretical Computer Science 4 (1) (1977) 1–22.
[6] G. D. Plotkin, Lcf considered as a programming language, Theoretical Computer Science 5 (3) (1977) 223–255.
[7] W. R. Cook, A denotational semantics of inheritance, Ph.D. thesis, Brown University (1989).
[8] A. Banerjee, D. A. Naumann, Ownership confinement ensures representation independence for object-oriented programs, Journal of the ACM 52 (6) (2005) 894–960.
[9] A. Banerjee, D. A. Naumann, State based ownership, reentrance, and encapsulation, in: A. P. Black (Ed.), ECOOP, Vol. 3586 of LNCS, Springer, 2005, pp. 387–411.
[10] A. Jeffrey, J. Rathke, Java Jr.: Fully abstract trace semantics for a core Java language, in: ESOP, 2005, pp. 423–438.
[11] M. Steffen, Object-connectivity and observability for class-based, object-oriented languages, Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel (Jul. 2006).
[12] E. Ábrahám, M. M. Bonsangue, F. S. de Boer, M. Steffen, Object connectivity and full abstraction for a concurrent calculus of classes, in: Z. Liu, K. Araki (Eds.), ICTAC 2004, Vol. 3407 of LNCS, Springer, Heidelberg, 2004, pp. 37–51.
[13] A. Gotsman, H. Yang, Liveness-preserving atomicity abstraction, in: L. Aceto, M. Henzinger, J. Sgall (Eds.), ICALP (2), Vol. 6756 of LNCS, Springer, 2011, pp. 453–465.
[14] I. Filipovic, P. W. O'Hearn, N. Rinetzky, H. Yang, Abstraction for concurrent objects, Theoretical Computer Science 411 (51-52) (2010) 4379–4398.
[15] M. Hennessy, R. Milner, On observing nondeterminism and concurrency., in: ICALP, 1980, pp. 299–309.
[16] E. Sumii, B. C. Pierce, A bisimulation for dynamic sealing, Theoretical Computer Science 375.
[17] E. Sumii, B. C. Pierce, A bisimulation for type abstraction and recursion, Journal of the ACM 54.
[18] V. Koutavas, M. Wand, Bisimulations for untyped imperative objects, in: P. Sestoft (Ed.), ESOP 2006, Vol. 3924 of LNCS, Springer, Heidelberg, 2006, pp. 146–161.
[19] V. Koutavas, M. Wand, Reasoning about class behavior, in: Informal Workshop Record of FOOL, 2007.
[20] N. Benton, Simple relational correctness proofs for static analyses and program transformations, in: N. D. Jones, X. Leroy (Eds.), POPL, ACM, 2004, pp. 14–25.
[21] S. Drossopoulou, A. Francalanza, P. Müller, A. Summers, A unified framework for verification techniques for object invariants, in: ECOOP, LNCS, 2008, pp. 412–437.
[22] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice-Hall, 1997.
[23] P. Müller, A. Poetzsch-Heffter, G. T. Leavens, Modular invariants for layered object structures, Science of Computer Programming. 62 (3) (2006) 253–286.
[24] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, W. Schulte, Verification of object-oriented programs with invariants, Journal of Object Technology 3 (6) (2004) 27–56.
[25] Y. Welsch, A. Poetzsch-Heffter, Full abstraction at package boundaries of object-oriented languages, in: SBMF 2011, LNCS, Springer, 2011, pp. 28–43.
[26] L. Lamport, How to write a proof, American Mathematical Monthly 102 (7) (1995) 600–608.
[27] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Third Edition, The Java Series, Addison-Wesley, Boston, Mass., 2005.
[28] C. Grothoff, J. Palsberg, J. Vitek, Encapsulating objects with confined types, in: OOPSLA, 2001, pp. 241–253.
[29] ECMA, C# Language Specification (Standard ECMA-334, 4th edition), `http://www.ecma-international.org/publications/standards/Ecma-334.htm`.
[30] F. Damiani, A. Poetzsch-Heffter, Y. Welsch, A type system for checking specialization of packages in object-oriented programming, to appear at SAC 2012.
[31] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (3) (2001) 396–450.
[32] A. K. Wright, M. Felleisen, A syntactic approach to type soundness, Inf. Comput. 115 (1) (1994) 38–94.
[33] J. H. Morris, Lambda-calculus models of programming languages, Tech. Rep. 57, MIT Laboratory for Computer Science (1968).
[34] M. Hennessy, Algebraic Theory of Processes, MIT Press, 1988.
[35] A. Jeffrey, J. Rathke, A fully abstract may testing semantics for concurrent objects, Theoretical Computer Science 338.
[36] C. A. R. Hoare, Proof of correctness of data representations, Acta Informatica 1 (1972) 271–281.
[37] C. C. Morgan, Programming from specifications, 2nd Edition, Prentice Hall International series in computer science, Prentice Hall, 1994.
[38] R.-J. J. Back, A. Akademi, J. V. Wright, Refinement Calculus: A Systematic Introduction, Springer, Heidelberg, 1998.
[39] K. Geilmann, A. Poetzsch-Heffter, Modular checking of confinement for object-oriented components using abstract interpretation, in: International Workshop on Aliasing, Confinement and Ownership, 2011.
[40] M. Abadi, K. R. M. Leino, A logic of object-oriented programs, in: N. Dershowitz (Ed.), Verification: Theory and Practice, Vol. 2772 of LNCS, Springer, Heidelberg, 2003, pp. 11–41.
[41] A. Poetzsch-Heffter, P. Müller, A programming logic for sequential java, in: S. D. Swierstra (Ed.), ESOP 1999, Vol. 1576 of LNCS, Springer, Heidelberg, 1999, pp. 162–176.
[42] J. Schäfer, A. Poetzsch-Heffter, JCobox: Generalizing active objects to concurrent components, in: T. D'Hondt (Ed.), ECOOP, Vol. 6183 of Lecture Notes in Computer Science, Springer, 2010, pp. 275–299.
[43] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A core language for abstract behavioral specification, in: B. Aichernig,

F. S. de Boer, M. M. Bonsangue (Eds.), Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010), Vol. 6957 of Lecture Notes in Computer Science, Springer-Verlag, 2011, pp. 142–164.

A. Additional Proofs

A.1. Proof of Theorem 1: Soundness and completeness of checkable conditions

Lemma A.1 (Completeness of checkable conditions)**.** *Consider two library implementations X and Y. If Y is source compatible with X then $S1_{X,Y}$-$S4_{X,Y}$ hold.*

PROOF: By contraposition:
ASSUME: $\vdash X$ and $\vdash Y$ such that $\mathrm{S1}_{X,Y}$-$\mathrm{S4}_{X,Y}$ do not hold
PROVE: $\exists K : \vdash KX$ and $\nvdash KY$
The proof considers each of the conditions on a per-case basis. For each condition, it assumes that the previous conditions hold. Let us consider a package name p_0 in the following which does neither occur in X nor Y.

⟨1⟩1. CASE: $\neg\mathrm{S1}_{X,Y}$
  ⟨2⟩1. CASE: $\exists p \in \mathcal{P}_Y \setminus \mathcal{P}_X$
    $\mathsf{uniquenames}_{KY}$ does not hold if $K \overset{\text{def}}{=}$ package p; public class c {}
  ⟨2⟩2. CASE: $\exists p \in \mathcal{P}_X \setminus \mathcal{P}_Y$
    ⟨3⟩1. $\exists t : p.t \in \mathcal{T}_X \wedge \mathsf{public}_K(p.t)$
      By **T-PACKAGE**
    ⟨3⟩2. $p.t \notin \mathcal{T}_Y$
      By ⟨2⟩2
    ⟨3⟩3. Q.E.D.
      T-CLASS ($\mathsf{acc}_{KY}(p.t,p_0)$) does not hold by ⟨3⟩2 if $K \overset{\text{def}}{=}$ package p_0; public class c { $p.t\ f$; }
⟨1⟩2. CASE: $\neg\mathrm{S2}_{X,Y}$
    LET: $p.t$ such that $\mathsf{public}_X(p.t) \wedge \neg\mathsf{public}_Y(p.t)$
  T-CLASS ($\mathsf{acc}_{KY}(p.t,p_0)$) does not hold if $K \overset{\text{def}}{=}$ package p_0; public class c { $p.t\ f$; }
⟨1⟩3. CASE: $\neg\mathrm{S3}_{X,Y}$
  ⟨2⟩1. CASE: $\mathsf{public}_X(p.t) \wedge \langle m, p_1.t_1 \cdot \overline{p_2.t_2}, \_\rangle \in_X p.t \wedge \langle m, p_1.t_1 \cdot \overline{p_2.t_2}, \_\rangle \notin_Y p.t$
    ⟨3⟩1. CASE: $\exists \overline{T} : \langle m, \overline{T}, \_\rangle \in_Y p.t \wedge \overline{T} \neq p_1.t_1 \cdot \overline{p_2.t_2}$
      ⟨4⟩1. CASE: $t = c$
        $\mathsf{C2}_{KY}$ does not hold if $K \overset{\text{def}}{=}$ package p_0; public class c_0 extds $p.c$ { $p_1.t_1\ m(\overline{p_2.t_2\ x})$ { null } }
      ⟨4⟩2. CASE: $t = i$
        $\mathsf{C2}_{KY}$ does not hold if $K \overset{\text{def}}{=}$ package p_0; public interface i_0 extds $p.i$ { $p_1.t_1\ m(\overline{p_2.t_2\ x})$; }
    ⟨3⟩2. CASE: $\nexists \overline{T} : \langle m, \overline{T}, \_\rangle \in_Y p.t$
      T-CALL does not hold if $K \overset{\text{def}}{=}$ package p_0; public class c_0 { lang.Object $m(p.t\ x)$ { $x.m(\overline{\mathsf{null}})$ } }
  ⟨2⟩2. CASE: $\mathsf{public}_X(p.t) \wedge \langle m, p_1.t_1 \cdot \overline{p_2.t_2}, \_\rangle \notin_X p.t \wedge \langle m, p_1.t_1 \cdot \overline{p_2.t_2}, \_\rangle \in_Y p.t$
    ⟨3⟩1. CASE: $\exists \overline{T} : \langle m, \overline{T}, \_\rangle \in_X p.t \wedge \overline{T} \neq p_1.t_1 \cdot \overline{p_2.t_2}$
      LET: $\overline{T} \overset{\text{def}}{=} p_3.t_3 \cdot \overline{p_4.t_4}$
      ⟨4⟩1. CASE: $t = c$
        $\mathsf{C2}_{KY}$ does not hold if $K \overset{\text{def}}{=}$ package p_0; public class c_0 extds $p.c$ { $p_3.t_3\ m(\overline{p_4.t_4\ x})$ { null } }
      ⟨4⟩2. CASE: $t = i$
        $\mathsf{C2}_{KY}$ does not hold if $K \overset{\text{def}}{=}$ package p_0; public interface i_0 extds $p.i$ { $p_3.t_3\ m(\overline{p_4.t_4\ x})$; }
    ⟨3⟩2. CASE: $\nexists \overline{T} : \langle m, \overline{T}, \_\rangle \in_X p.t$
      ⟨4⟩1. CASE: $t = c$
        $\mathsf{C2}_{KY}$ does not hold if $K \overset{\text{def}}{=}$ package p_0; public class c_0 extds $p.c$ { $p_0.c_0\ m()$ { null } }
      ⟨4⟩2. CASE: $t = i$
        $\mathsf{C2}_{KY}$ does not hold if $K \overset{\text{def}}{=}$ package p_0; public interface i_0 extds $p.i$ { $p_0.c_0\ m()$; }
⟨1⟩4. CASE: $\neg\mathrm{S4}_{X,Y}$
    LET: $p_1.t_1, p_2.t_2$ such that $\mathsf{public}_X(p_1.t_1) \wedge \mathsf{public}_X(p_2.t_2) \wedge p_1.t_1 \leq_X p_2.t_2 \wedge p_1.t_1 \nleq_Y p_2.t_2$
  T-SUB does not hold if $K \overset{\text{def}}{=}$ package p_0; public class c { $p_2.t_2\ m(p_1.t_1\ x)$ { x } } □

Lemma A.2 (Soundness of checkable conditions)**.** *Consider two library implementations X and Y. If $S1_{X,Y}$-$S4_{X,Y}$ hold, then Y is source compatible with X.*

PROOF: Direct proof:
ASSUME: $\vdash X$ and $\vdash Y$ and $\mathrm{S1}_{X,Y}$-$\mathrm{S4}_{X,Y}$ hold and $\vdash KX$
PROVE: $\vdash KY$
⟨1⟩1. $p \in \mathcal{P}_K \wedge \mathsf{acc}_{KX}(p_1.t_1, p) \rightarrow \mathsf{acc}_{KY}(p_1.t_1, p)$
  Directly from Def. of acc and $\mathrm{S2}_{X,Y}$
⟨1⟩2. $p.t \in \mathcal{T}_K \rightarrow (\langle m, \overline{T}, \_\rangle \in_{KX} p.t \leftrightarrow \langle m, \overline{T}, \_\rangle \in_{KY} p.t)$
  We show one direction, the other is similar:
  ⟨2⟩1. CASE: $p.t \in \mathcal{T}_K \wedge \langle m, \overline{T}, p.t\rangle \in_{KX} p.t$
    Trivial
  ⟨2⟩2. CASE: $p.t \in \mathcal{T}_K \wedge \langle m, \overline{T}, p.t\rangle \notin_{KX} p.t$
    Then $\langle m, \overline{T}, p.t\rangle \notin_{KY} p.t$. We apply D-INH-METHOD-C or D-INH-METHOD-I:
    ⟨3⟩1. CASE: $p.t <^{\mathbf{d}}_{KX} p_0.t_0 \wedge p_0.t_0 \in \mathcal{T}_K$
      Apply again ⟨2⟩1 or ⟨2⟩2 (as $<_{KX}$ acyclic)
    ⟨3⟩2. CASE: $p.t <^{\mathbf{d}}_{KX} p_0.t_0 \wedge p_0.t_0 \in \mathcal{T}_X$
      By $\mathrm{S3}_{X,Y}$
⟨1⟩3. $\mathsf{acc}_{KX}(p_1.t_1, p) \wedge \mathsf{acc}_{KX}(p_2.t_2, p) \wedge p_1.t_1 \leq_{KX} p_2.t_2 \rightarrow p_1.t_1 \leq_{KY} p_2.t_2$
  By $\mathrm{S2}_{X,Y}$ and $\mathrm{S4}_{X,Y}$
⟨1⟩4. Consider $p.c \in \mathcal{C}_K$ and $\mathsf{acc}_{KX}(\mathsf{rng}(\Gamma), p)$. If $KX, p.c, \Gamma \vdash E : T$ then $KY, p.c, \Gamma \vdash E : T$. Similarly, if $KX, p.c, \Gamma \vdash E :_{\leq} T$ then $KY, p.c, \Gamma \vdash E :_{\leq} T$.
  PROOF: By induction on the typing derivation of E:
  ⟨2⟩1. Induction basis

$\langle 3\rangle 1$. Case: **T-Null** and **T-Var**
Trivial
$\langle 3\rangle 2$. Case: **T-New**
By $\langle 1\rangle 1$
$\langle 2\rangle 2$. Induction step
$\langle 3\rangle 1$. Case: **T-Get**, **T-Set** and **T-Let**
Trivial
$\langle 3\rangle 2$. Case: **T-Sub**
By $\langle 1\rangle 1$ and $\langle 1\rangle 3$
$\langle 3\rangle 3$. Case: **T-If** and **T-Cast**
By $\langle 1\rangle 3$ and Def. of $\mathsf{cmp}(,)$
$\langle 3\rangle 4$. Case: **T-Call**
By $\langle 1\rangle 2$ and $S3_{X,Y}$
$\langle 1\rangle 5$. $\vdash KY$
Proof: By induction on the typing derivation of $\vdash KY$:
$\langle 2\rangle 1$. **T-Lib**
$\langle 3\rangle 1$. $\mathsf{uniquenames}_{KY}$
Trivial
$\langle 3\rangle 2$. $<_{KY}$ acyclic
By $<_{KX}$ and $<_{Y}$ acyclic and $\vdash Y$
$\langle 3\rangle 3$. $C1_{KY}$
By $S2_{X,Y}$
$\langle 3\rangle 4$. $C2_{KY}$-$C4_{KY}$
By $S2_{X,Y}$-$S4_{X,Y}$ and $\langle 1\rangle 2$
$\langle 2\rangle 2$. **T-Package** and **T-Intf**
Trivial
$\langle 2\rangle 3$. **T-Class** and **T-MethSig**
By $\langle 1\rangle 1$
$\langle 2\rangle 4$. **T-Method**
By $\langle 1\rangle 4$ □

A.2. Proof of Theorem 2: Full abstraction

We first state two helper lemmas.

Lemma A.3 (Well-formed traces). *Let K be a program context of the library implementation X. If $\overline{\gamma} \in \mathsf{traces}(KX)$, then $\overline{\gamma}$ is of the form $(\mu\downarrow \cdot \mu\uparrow)^{*} \cdot (\mathsf{succ} \mid \mu\downarrow)^{?}$.*

Proof: Follows directly from the operational rules. □

Lemma A.4 (Abstract behavior). *Let Y be source compatible with X, K a deterministic program context of X and $\overline{\gamma} \in \mathsf{traces}(KX)$. If $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$, then $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$.*

Proof: By induction on the length of $\overline{\gamma}$
$\langle 1\rangle 1$. Induction basis
By **L-Empty**, we have $\mathcal{S}^{\mathbf{init}}_{KY} \xrightarrow{\bullet} \mathcal{S}^{\mathbf{init}}_{KY}$
$\langle 1\rangle 2$. Induction step
Assume: a) $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(KX)$
b) $\overline{\gamma} \cdot \gamma \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$
Prove: $\overline{\gamma} \cdot \gamma \in \mathsf{abs}_X(\mathsf{traces}(KY))$
$\langle 2\rangle 1$. $\overline{\gamma} \in \mathsf{traces}(KX)$
By $\langle 1\rangle 2$-a and prefix-closedness of Definition 5.4
$\langle 2\rangle 2$. $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$
By $\langle 1\rangle 2$-b and prefix-closedness of Definition 5.4
$\langle 2\rangle 3$. $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$
By induction hypothesis from $\langle 2\rangle 1$ and $\langle 2\rangle 2$
$\langle 2\rangle 4$. Case: $\overline{\gamma} = \bullet$ or $\mathsf{last}(\overline{\gamma}) = \mu\uparrow$
From Lemma 6.2 by $\langle 2\rangle 1$, $\langle 2\rangle 3$, $\langle 1\rangle 2$-a and $\langle 2\rangle 4$
$\langle 2\rangle 5$. Case: $\mathsf{last}(\overline{\gamma}) = \mu\downarrow$
From Lemma 6.1 by $\langle 2\rangle 2$, $\langle 2\rangle 3$, $\langle 1\rangle 2$-b and $\langle 2\rangle 5$
$\langle 2\rangle 6$. Q.E.D.
Cases exhaustive due to $\langle 1\rangle 2$-a and Lemma A.3 □

Theorem 2 (Full abstraction). *Consider two library implementations X and Y. Then Y is trace compatible with X iff Y is contextually compatible with X.*

Proof: We show both directions:
$\langle 1\rangle 1$. Case: Trace implies contextual compatibility
Assume: a) $\mathsf{traces}(\mathsf{mgc}(X)X) \subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$
b) K a deterministic program context of X such that $\mathcal{S}^{\mathbf{init}}_{KX}\checkmark$
Prove: $\mathcal{S}^{\mathbf{init}}_{KY}\checkmark$
$\langle 2\rangle 1$. There is $\overline{\gamma} \in \mathsf{traces}(KX)$ such that $\mathsf{last}(\overline{\gamma}) = \mathsf{succ}$
From Definition 6.1 by $\langle 1\rangle 1$-b
$\langle 2\rangle 2$. $\overline{\gamma} \in \mathsf{traces}(\mathsf{mgc}(X)X)$
From Lemma 6.3 by $\langle 2\rangle 1$
$\langle 2\rangle 3$. $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$
By $\langle 2\rangle 2$ and $\langle 1\rangle 1$-a
$\langle 2\rangle 4$. $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$
From Lemma A.4 by $\langle 2\rangle 1$ and $\langle 2\rangle 3$
$\langle 2\rangle 5$. Q.E.D.
From Definition 6.1 by $\langle 2\rangle 1$ and $\langle 2\rangle 4$
$\langle 1\rangle 2$. Case: Testing implies trace compatibility
Instead of proving the stronger property $\mathsf{traces}(\mathsf{mgc}(X)X) \subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$, we prove that $\mathsf{traces}(\mathsf{mgc}(X)X) \subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$ $\star$. The claim then follows directly from Lemma 6.5. The proof of $\star$ goes by contraposition:
Assume: $\mathsf{traces}(\mathsf{mgc}(X)X) \not\subseteq \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$
Prove: Y is not contextually compatible with X
This is equivalent to the following formulation, as the empty trace is in both sets and they are prefix-closed:
Assume: $\overline{\gamma} \cdot \gamma$ such that
a) $\overline{\gamma} \in \mathsf{traces}(\mathsf{mgc}(X)X)$

b) $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$
c) $\overline{\gamma} \cdot \gamma \in \mathsf{traces}(\mathsf{mgc}(X)X)$
d) $\overline{\gamma} \cdot \gamma \notin \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$

PROVE: There is a deterministic program context K such that $\mathcal{S}^{\mathbf{init}}_{KX}$✓ and $\mathcal{S}^{\mathbf{init}}_{KY}$✗

⟨2⟩1. $\mathsf{last}(\overline{\gamma}) = \mu$📥
  ⟨3⟩1. $\overline{\gamma} \neq \bullet$ and $\mathsf{last}(\overline{\gamma}) \neq \mu$📤
    ASSUME: $\overline{\gamma} = \bullet$ or $\mathsf{last}(\overline{\gamma}) = \mu$📤
    PROVE: Contradiction
    ⟨4⟩1. $\overline{\gamma} \cdot \gamma \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$
      From Lemma 6.2 by ⟨1⟩2-a, ⟨1⟩2-b and ⟨1⟩2-c
    ⟨4⟩2. Q.E.D.
      By ⟨1⟩2-d and ⟨4⟩1
  ⟨3⟩2. $\mathsf{last}(\overline{\gamma}) \neq \mathsf{succ}$
    From Lemma A.3 by ⟨1⟩2-c
  ⟨3⟩3. Q.E.D.
    By ⟨3⟩1 and ⟨3⟩2
⟨2⟩2. $\gamma = \mu$📤
  From Lemma A.3 by ⟨2⟩1 and ⟨1⟩2-c
⟨2⟩3. CASE: $\exists \gamma'$ such that $\overline{\gamma} \cdot \gamma' \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$ and $\overline{\gamma} \cdot \gamma' \not\equiv \overline{\gamma} \cdot \gamma$
  From Lemma 6.6 by ⟨1⟩2-a, ⟨1⟩2-b, ⟨1⟩2-c, ⟨2⟩3 and ⟨2⟩1
⟨2⟩4. CASE: $\nexists \gamma'$ such that $\overline{\gamma} \cdot \gamma' \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$
  ⟨3⟩1. $\overline{\gamma} \cdot \gamma \cdot \mathsf{succ} \in \mathsf{traces}(\mathsf{mgc}(X)X)$
    From Lemma 6.7 by ⟨1⟩2-a and ⟨2⟩2
  ⟨3⟩2. Let K be the deterministic program context such that $\overline{\gamma} \cdot \gamma \cdot \mathsf{succ} \in \mathsf{traces}(KX)$
    From Lemma 6.4 by ⟨3⟩1
  ⟨3⟩3. $\mathcal{S}^{\mathbf{init}}_{KX}$✓
    From Definition 6.1 by ⟨3⟩2
  ⟨3⟩4. $\mathcal{S}^{\mathbf{init}}_{KY}$✗
    ⟨4⟩1. $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(KY))$
      ⟨5⟩1. $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$
        From Lemma 6.5 by ⟨1⟩2-b
      ⟨5⟩2. Q.E.D.
        From Lemma A.4 by ⟨3⟩2 and ⟨5⟩1
    ⟨4⟩2. $\nexists \gamma' : \overline{\gamma} \cdot \gamma' \in \mathsf{abs}_X(\mathsf{traces}(KY))$
      ⟨5⟩1. $\nexists \gamma' : \overline{\gamma} \cdot \gamma' \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$
        By contradiction:
        ASSUME: $\overline{\gamma} \cdot \gamma' \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$
        PROVE: $\overline{\gamma} \cdot \gamma' \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(X)Y))$ which contradicts ⟨2⟩4
        ⟨6⟩1. $\overline{\gamma} \in \mathsf{abs}_X(\mathsf{traces}(\mathsf{mgc}(Y)Y))$
          By assumption ⟨5⟩1 and prefix-closedness of Definition 5.4
        ⟨6⟩2. Q.E.D.
          From Lemma 6.1 by ⟨1⟩2-b, ⟨6⟩1 and ⟨2⟩1
      ⟨5⟩2. Q.E.D.
        From Lemma 6.3 by ⟨5⟩1
    ⟨4⟩3. Q.E.D.
      From Definition 6.1 by ⟨4⟩1, ⟨4⟩2 and ⟨2⟩1
  ⟨3⟩5. Q.E.D.
    By ⟨3⟩2, ⟨3⟩3 and ⟨3⟩4
⟨2⟩5. Q.E.D.
  Cases exhaustive due to ⟨1⟩2-d □

A.3. Proofs for the relations $\preccurlyeq_L$

Lemma A.5 (Similar large step from L preserves $\preccurlyeq_L$). *If $\mathcal{S}_1 \preccurlyeq^{\rho}_{L} \mathcal{S}_2$ and $\mathcal{S}_1 \xrightarrow{\gamma_1} \mathcal{S}'_1$ and $\mathsf{exec}(\mathcal{S}_1) = L$ and $\mathcal{S}_2 \xrightarrow{\gamma_2} \mathcal{S}'_2$ and $\gamma_1 \equiv^{\rho^{12}_{\gamma}} \mathsf{abs}_{X_1}(\gamma_2)$ and ρ^{12}_{γ} minimal and consistent with ρ, then $\exists \mathcal{S}'_3$ such that $\mathcal{S}_2 \xrightarrow{\gamma_3} \mathcal{S}'_3$ and $\gamma_1 \equiv^{\rho^{13}_{\gamma}} \mathsf{abs}_{X_1}(\gamma_3)$ and ρ^{13}_{γ} minimal and consistent with ρ and $\mathcal{S}'_1 \preccurlyeq^{\rho \cup \rho^{13}_{\gamma}}_{L} \mathcal{S}'_3$ and $\mathcal{S}'_2 \preccurlyeq^{\rho^{id}_{\mathcal{S}_2} \cup (\rho^{13}_{\gamma} \circ \rho^{12^{-1}}_{\gamma})}_{\neg L} \mathcal{S}'_3$.*

PROOF:
ASSUME: a) $\mathcal{S}_1 \preccurlyeq^{\rho}_{L} \mathcal{S}_2$
b) $\mathcal{S}_1 \xrightarrow{l_1} \mathcal{S}'_1$
c) $\mathsf{exec}(\mathcal{S}_1) = L$
d) $\mathcal{S}_2 \xrightarrow{l_2} \mathcal{S}'_2$
e) $l_1 \equiv^{\rho^{12}_{l}} \mathsf{abs}_{X_1}(l_2)$ and ρ^{12}_{l} minimal
f) ρ^{12}_{l} consistent with ρ

⟨1⟩1. $\exists \mathcal{S}'_3$ such that $\mathcal{S}_2 \xrightarrow{l_3} \mathcal{S}'_3$ and $l_1 \equiv^{\rho^{13}_{l}} \mathsf{abs}_{X_1}(l_3)$ and ρ^{13}_{l} minimal and consistent with ρ and $\mathcal{S}'_1 \preccurlyeq^{\rho \cup \rho^{13}_{l}}_{L} \mathcal{S}'_3$
  From Lemma 7.8 by ⟨⟩-a, ⟨⟩-b and ⟨⟩-c
⟨1⟩2. $\mathcal{S}'_2 \preccurlyeq^{\rho^{id}_{\mathcal{S}_2} \cup (\rho^{13}_{l} \circ \rho^{12^{-1}}_{l})}_{\neg L} \mathcal{S}'_3$
  ⟨2⟩1. $l_1 \equiv^{\rho^{13}_{l} \circ \rho^{12^{-1}}_{l}} \mathsf{abs}_{X_1}(l_2)$ where $\rho^{13}_{l} \circ \rho^{12^{-1}}_{l}$ minimal
  ⟨2⟩2. $\mathcal{S}_2 \preccurlyeq^{\rho^{id}_{\mathcal{S}_2}}_{\neg L} \mathcal{S}_2$
  ⟨2⟩3. $\rho^{13}_{l} \circ \rho^{12^{-1}}_{l}$ consistent with $\rho \circ \rho^{-1}$
  ⟨2⟩4. $\rho \circ \rho^{-1} = \rho^{id}_{\mathcal{S}_2}$
  ⟨2⟩5. Q.E.D.
    From Lemma 7.9 by previous steps
⟨1⟩3. Q.E.D.
  By previous steps □

Lemma A.6 (Multiple large steps preserve $\preccurlyeq_L$). *If $\mathcal{S}_1 \preccurlyeq^{\rho}_{L} \mathcal{S}_2$ and $\mathcal{S}_1 \xrightarrow{\overline{\gamma}_1} \mathcal{S}'_1$ and $\mathcal{S}_2 \xrightarrow{\overline{\gamma}_2} \mathcal{S}'_2$ and $\overline{\gamma}_1 \equiv^{\rho^{12}_{\overline{\gamma}}} \mathsf{abs}_{X_1}(\overline{\gamma}_2)$ and $\rho^{12}_{\overline{\gamma}}$ minimal and consistent with ρ, then $\exists \mathcal{S}'_3$ such that $\mathcal{S}_2 \xrightarrow{\overline{\gamma}_3} \mathcal{S}'_3$ and $\overline{\gamma}_1 \equiv^{\rho^{13}_{\overline{\gamma}}} \mathsf{abs}_{X_1}(\overline{\gamma}_3)$ and $\rho^{13}_{\overline{\gamma}}$ minimal and consistent with ρ and $\mathcal{S}'_1 \preccurlyeq^{\rho \cup \rho^{13}_{\overline{\gamma}}}_{L} \mathcal{S}'_3$ and $\mathcal{S}'_2 \preccurlyeq^{\rho^{id}_{\mathcal{S}_2} \cup (\rho^{13}_{\overline{\gamma}} \circ \rho^{12^{-1}}_{\overline{\gamma}})}_{\neg L} \mathcal{S}'_3$.*

PROOF: By induction on the length of trace $\overline{\gamma}_1$

$\langle 1\rangle 1.$ Induction basis: empty trace $\overline{\gamma}_1 = \bullet$

Trivial

$\langle 1\rangle 2.$ Induction step

ASSUME: a) $\mathcal{S}_1 \preccurlyeq_L^{\rho} \mathcal{S}_2$

b) $\mathcal{S}_1 \xrightarrow{\overline{\gamma}_1} \mathcal{S}_1' \xrightarrow{\gamma_1} \mathcal{S}_1''$

c) $\mathcal{S}_2 \xrightarrow{\overline{\gamma}_2} \mathcal{S}_2' \xrightarrow{\gamma_2} \mathcal{S}_2''$

d) $\overline{\gamma}_1 \cdot \gamma_1 \equiv^{\rho^{12}_{\overline{\gamma}\gamma}} \mathsf{abs}_{X_1}(\overline{\gamma}_2 \cdot \gamma_2)$ and $\rho^{12}_{\overline{\gamma}\gamma}$ minimal

e) $\rho^{12}_{\overline{\gamma}\gamma}$ consistent with ρ

PROVE: $\exists \mathcal{S}_3''$ such that

a) $\mathcal{S}_2 \xrightarrow{\overline{\gamma}_3 \cdot \gamma_3} \mathcal{S}_3''$ and

b) $\overline{\gamma}_1 \cdot \gamma_1 \equiv^{\rho^{13}_{\overline{\gamma}\gamma}} \mathsf{abs}_{X_1}(\overline{\gamma}_3 \cdot \gamma_3)$ and

c) $\rho^{13}_{\overline{\gamma}\gamma}$ minimal and consistent with ρ and

d) $\mathcal{S}_1'' \preccurlyeq_L^{\rho \cup \rho^{13}_{\overline{\gamma}\gamma}} \mathcal{S}_3''$ and

e) $\mathcal{S}_2'' \preccurlyeq_{\neg L}^{\rho^{id}_{\mathcal{S}_2} \cup (\rho^{13}_{\overline{\gamma}\gamma} \circ \rho^{12^{-1}}_{\overline{\gamma}\gamma})} \mathcal{S}_3''$

$\langle 2\rangle 1.$ By induction hypothesis we get $\mathcal{S}_3'$ such that

a) $\mathcal{S}_2 \xrightarrow{\overline{\gamma}_3} \mathcal{S}_3'$

b) $\mathcal{S}_1' \preccurlyeq_L^{\rho \cup \rho^{13}_{\overline{\gamma}}} \mathcal{S}_3'$

c) $\mathcal{S}_2' \preccurlyeq_{\neg L}^{\rho^{id}_{\mathcal{S}_2} \cup (\rho^{13}_{\overline{\gamma}} \circ \rho^{12^{-1}}_{\overline{\gamma}})} \mathcal{S}_3'$

d) $\overline{\gamma}_1 \equiv^{\rho^{13}_{\overline{\gamma}}} \mathsf{abs}_{X_1}(\overline{\gamma}_3)$ and $\rho^{13}_{\overline{\gamma}}$ minimal

e) $\rho^{13}_{\overline{\gamma}}$ consistent with ρ

$\langle 2\rangle 2.$ CASE: $\mathsf{exec}(\mathcal{S}_1') = L$

$\langle 3\rangle 1.$ By $\langle 1\rangle$2-b, $\langle 2\rangle$1-b and $\langle 2\rangle$2, Lemma 7.8 yields $\mathcal{S}_3''$ such that

a) $\mathcal{S}_3' \xrightarrow{\gamma_3} \mathcal{S}_3''$

b) $\mathcal{S}_1'' \preccurlyeq_L^{\rho \cup \rho^{13}_{\overline{\gamma}} \cup \rho^{13}_{\gamma}} \mathcal{S}_3''$

c) $\gamma_1 \equiv^{\rho^{13}_{\gamma}} \mathsf{abs}_{X_1}(\gamma_3)$ and ρ^{13}_{γ} minimal

d) ρ^{13}_{γ} consistent with $\rho \cup \rho^{13}_{\overline{\gamma}}$

$\langle 3\rangle 2.$ ρ^{13}_{γ} consistent with $\rho^{13}_{\overline{\gamma}}$

$\langle 3\rangle 3.$ $\overline{\gamma}_1 \cdot \gamma_1 \equiv^{\rho^{13}_{\overline{\gamma}\gamma}} \mathsf{abs}_{X_1}(\overline{\gamma}_3 \cdot \gamma_3)$ and $\rho^{13}_{\overline{\gamma}\gamma} \overset{\text{def}}{=} \rho^{13}_{\overline{\gamma}} \cup \rho^{13}_{\gamma}$ minimal

$\langle 3\rangle 4.$ $\rho^{13}_{\overline{\gamma}\gamma}$ consistent with ρ

By step $\langle 3\rangle$3 and $\langle 3\rangle$1-d

$\langle 3\rangle 5.$ $\mathcal{S}_2'' \preccurlyeq_{\neg L}^{\rho^{id}_{\mathcal{S}_2} \cup (\rho^{13}_{\overline{\gamma}\gamma} \circ \rho^{12^{-1}}_{\overline{\gamma}\gamma})} \mathcal{S}_3''$

$\langle 4\rangle 1.$ There are unique $\rho^{12}_{\overline{\gamma}}$ and ρ^{12}_{γ} such that $\overline{\gamma}_1 \equiv^{\rho^{12}_{\overline{\gamma}}} \mathsf{abs}_{X_1}(\overline{\gamma}_2)$ and $\gamma_1 \equiv^{\rho^{12}_{\gamma}} \mathsf{abs}_{X_1}(\gamma_2)$ and $\rho^{12}_{\overline{\gamma}}, \rho^{12}_{\gamma}$ minimal and $\rho^{12}_{\overline{\gamma}\gamma} = \rho^{12}_{\overline{\gamma}} \cup \rho^{12}_{\gamma}$ and $\rho^{12}_{\overline{\gamma}}$ consistent with ρ^{12}_{γ}

$\langle 4\rangle 2.$ $\overline{\gamma}_2 \equiv^{\rho^{13}_{\overline{\gamma}} \circ \rho^{12^{-1}}_{\overline{\gamma}}} \overline{\gamma}_3$ where $\rho^{13}_{\overline{\gamma}} \circ \rho^{12^{-1}}_{\overline{\gamma}}$ minimal

$\langle 4\rangle 3.$ $\gamma_2 \equiv^{\rho^{13}_{\gamma} \circ \rho^{12^{-1}}_{\gamma}} \gamma_3$ where $\rho^{13}_{\gamma} \circ \rho^{12^{-1}}_{\gamma}$ minimal

$\langle 4\rangle 4.$ ρ^{12}_{γ} consistent with $\rho \cup \rho^{12}_{\overline{\gamma}}$

By $\langle 1\rangle$2-e

$\langle 4\rangle 5.$ $\rho^{13}_{\gamma} \circ \rho^{12^{-1}}_{\gamma}$ consistent with $\rho^{id}_{\mathcal{S}_2} \cup (\rho^{13}_{\overline{\gamma}} \circ \rho^{12^{-1}}_{\overline{\gamma}})$

$\langle 5\rangle 1.$ $\rho^{13}_{\gamma} \circ \rho^{12^{-1}}_{\gamma}$ consistent with $(\rho \cup \rho^{13}_{\overline{\gamma}}) \circ (\rho \cup \rho^{12}_{\overline{\gamma}})^{-1}$

$\langle 5\rangle 2.$ $(\rho \cup \rho^{12}_{\overline{\gamma}})^{-1} = \rho^{-1} \cup \rho^{12^{-1}}_{\overline{\gamma}}$

$\langle 5\rangle 3.$ $(\rho \cup \rho^{13}_{\overline{\gamma}}) \circ (\rho^{-1} \cup \rho^{12^{-1}}_{\overline{\gamma}}) = (\rho \circ \rho^{-1}) \cup (\rho^{13}_{\overline{\gamma}} \circ \rho^{12^{-1}}_{\overline{\gamma}})$

$\langle 5\rangle 4.$ $(\rho \circ \rho^{-1}) = \rho^{id}_{\mathcal{S}_2}$

$\langle 5\rangle 5.$ Q.E.D.

By $\langle 5\rangle$1 and the equalities $\langle 5\rangle$2, $\langle 5\rangle$3 and $\langle 5\rangle$4

$\langle 4\rangle 6.$ $\mathcal{S}_2'' \preccurlyeq_{\neg L}^{\rho^{id}_{\mathcal{S}_2} \cup (\rho^{13}_{\overline{\gamma}} \circ \rho^{12^{-1}}_{\overline{\gamma}}) \cup (\rho^{13}_{\gamma} \circ \rho^{12^{-1}}_{\gamma})} \mathcal{S}_3''$

From Lemma 7.9 by $\langle 1\rangle$2-c and $\langle 3\rangle$1-a, $\langle 4\rangle$3 and $\langle 4\rangle$5

$\langle 4\rangle 7.$ $\rho^{13}_{\overline{\gamma}\gamma} \circ \rho^{12^{-1}}_{\overline{\gamma}\gamma} = (\rho^{13}_{\overline{\gamma}} \circ \rho^{12^{-1}}_{\overline{\gamma}}) \cup (\rho^{13}_{\gamma} \circ \rho^{12^{-1}}_{\gamma})$

$\langle 5\rangle 1.$ $\rho^{13}_{\overline{\gamma}\gamma} \circ \rho^{12^{-1}}_{\overline{\gamma}\gamma} = (\rho^{13}_{\overline{\gamma}} \cup \rho^{13}_{\gamma}) \circ (\rho^{12}_{\overline{\gamma}} \cup \rho^{12}_{\gamma})^{-1}$

By $\langle 3\rangle$3 and $\langle 4\rangle$1

$\langle 5\rangle 2.$ $(\rho^{13}_{\overline{\gamma}} \cup \rho^{13}_{\gamma}) \circ (\rho^{12}_{\overline{\gamma}} \cup \rho^{12}_{\gamma})^{-1} = (\rho^{13}_{\overline{\gamma}} \cup \rho^{13}_{\gamma}) \circ (\rho^{12^{-1}}_{\overline{\gamma}} \cup \rho^{12^{-1}}_{\gamma})$

$\langle 5\rangle 3.$ $(\rho^{13}_{\overline{\gamma}} \cup \rho^{13}_{\gamma}) \circ (\rho^{12^{-1}}_{\overline{\gamma}} \cup \rho^{12^{-1}}_{\gamma}) = (\rho^{13}_{\overline{\gamma}} \circ \rho^{12^{-1}}_{\overline{\gamma}}) \cup (\rho^{13}_{\gamma} \circ \rho^{12^{-1}}_{\gamma})$

$\langle 5\rangle 4.$ Q.E.D.

By previous equalities

$\langle 4\rangle 8.$ Q.E.D.

By $\langle 4\rangle$6 and equality $\langle 4\rangle$7

$\langle 3\rangle 6.$ Q.E.D.

By $\langle 2\rangle$1-a, $\langle 3\rangle$1-a, $\langle 3\rangle$1-b, $\langle 3\rangle$3, $\langle 3\rangle$4 and $\langle 3\rangle$5

$\langle 2\rangle 3.$ CASE: $\mathsf{exec}(\mathcal{S}_1') = \neg L$

Similar to the previous case

$\langle 2\rangle 4.$ Q.E.D.

Cases are exhaustive □