Safe Uniform Proxies for Java

Patrick Eugster

Department of Computer Science
Purdue University
West Lafayette 47906 IN, USA

Abstract

The proxy abstraction has a long-lasting tradition in object-oriented programming. From design patterns to inherent programming language support, from remote method invocations to simple forms of behavioral reflection – incarnations as well as applications of proxies are innumerable.

Since version 1.3, Java supports the concept of dynamic proxy. Such an object conforms to a set of types specified by the program and can be used wherever an expression of any of these types is expected, yet reifies invocations performed on it. This ability has allowed dynamic proxies to be used to implement paradigms such as behavioral reflection, structural conformance, or multi-methods. Alas, these proxies are only available “for interfaces”. The case of creating dynamic proxies for a set of types including a class type has not been addressed, meaning that it is currently not possible to create a dynamic proxy that conforms to an application-defined class type. This weakness strongly limits any application of dynamic proxies beyond the inherent limitations of proxies, which have motivated deeper programming language support for features such as behavioral reflection.

In this paper we unfold the current support for dynamic proxies in Java, assessing it in the light of a set of generic criteria for proxy implementations. We present an approach to supporting dynamic proxies “for classes”, consisting in transformations performed on classes at load-time, including a generic scheme for enforcing encapsulation upon field accesses. These transformations seamlessly extend the scope of the current support for dynamic proxies from the programmer’s perspective. We argue for the safety of our transformations, and discuss the precise benefits and costs of our extension in terms of the criteria introduced through several application scenarios for dynamic proxies, including an implementation of future method invocations balancing safety and transparency. We further dissect transparency, identifying application scenarios which benefit most from proxies and which conversely are better supported by language features or design patterns.

Email address: p@cs.purdue.edu (Patrick Eugster)

This article builds on material which appeared in the proceedings of OOPSLA’06.
1. Introduction

The concept of proxy has a longlasting tradition in object-oriented programming, enjoying both innumerable incarnations and applications. The proxy design pattern and its relatives, such as the decorator pattern (responsibilities can be dynamically “attached” to objects) or the adapter pattern (method invocations performed on an expression can be “translated”) [23], for instance, are probably among the most prominent of all design patterns. Examples of widespread and traditional applications of proxies are remote method invocations [56], future objects in asynchronous so-called future method invocations [73], and behavioral reflection [38].

1.1. Java Dynamic Proxies

At version 1.3, Java™’s core reflection API [63] has seen the addition of dynamic proxies. A dynamic proxy is a typed proxy, created at run-time for a type (more generally a set of types) defined by the application. Such an object can be used in a consistent manner wherever an expression of that type (of any of those types) is expected. The invocation of a method $m$ on such a dynamic proxy object is however reified, somehow stepping from a statically typed context to dynamic interaction where any action can be performed in the confines of $m$’s execution. Together with dynamic invocation facilities introduced earlier in Java as part of its introspection features, the concept of dynamic proxy enables graceful realizations of the above-mentioned patterns and applications. Examples are plentiful. Various reports can be found in the world wide web on implementations of popular paradigms based on Java’s dynamic proxies, e.g., implicit (structural) conformance, future invocations, dynamic multi-dispatch (a.k.a. multi-methods), design by contract [47] or aspect-oriented programming [39] (see for instance [16] and [34] respectively for the latter two). Furthermore, dynamic proxies have replaced the rmic pre-compiler for creating proxies for Java RMI.

1.2. Limitations

The implementation of dynamic proxies in Java is simple and elegant. When creating a dynamic proxy for an interface $I$, an instance of a class implementing $I$ is created, that class being generated automatically as byte-code at run-time, loaded, and linked. This requires no specific support from the Java compiler or virtual machine [62]. Leaving aside inherent limitations of proxies when using them specifically for “mimicking” other objects (a proxy and the object it represents remain distinct entities), this simple solution unfortunately manifests important limitations: dynamic proxies are not uniformly available, but only “for interfaces”. More precisely, while dynamic proxies can be created even to implement several interfaces, such proxies can not be assigned to variables whose static type is a class type. This limitation strongly hampers the potential of dynamic proxies. To fully exploit the above-mentioned paradigms through implementations based on dynamic proxies, programs are ultimately constrained to define all variables as being of interface types, and to use classes only for instantiation. Take the case of future invocations, which intuitively makes an ideal case for dynamic proxies. Unless respecting the above-mentioned constraint, they can currently only be implemented in an explicit form [73] without such proxies. Implicit, more transparent futures can be achieved of course by other means. Praktikakis et al. [52] for instance
make use of a static analysis and corresponding program transformations, which are however specific to future method invocations.

1.3. Contributions

The goal of this paper is to make the concept of dynamic proxies in Java more uniformly available, i.e., also “for classes”\footnote{We refer to such proxies as “proxies for classes” for simplicity, though it is an abuse of language as we are not concerned with creating single proxies for entire classes.}. After unfoldling the current support for dynamic proxies and its implementation in Java we make the following contributions:

- Limitations of proxies are discussed in general and in the case of Java, introducing a set of criteria to evaluate a "proxification" scheme. This set includes dimensions such as safety, security, transparency, or performance overhead.

- We propose uniform dynamic proxies for Java, building on the existing support for dynamic proxies in Java. This leads to creating a dynamic proxy class \(C\) for a set of types including a class \(C'\), as a subclass of \(C'\) itself. In order to make such an extension approach feasible, we propose a set of modular byte-code transformations including a general scheme for transforming instance field accesses to invocations of automatically generated getter/setter methods. We formalize this transformation together with the current support for dynamic proxies as an extension to Featherweight Java [33], illustrating safety for our extension in the presence of field hiding. In contrast similar transformations described in the literature are unsafe in that they do not capture field hiding and thus homonymous fields in a class and its super-class can be collapsed, or suffer from similar issues.

- Our transformations yield an internal uniformly virtual object model, in which any type/member can be extended/overridden. We present how we have tamed the power of this model in the code loading process to ensure original extension and visibility constraints of programs. We report on measurements conducted with the SpecJVM benchmark of the overhead introduced by our object model.

- The benefits and limitations of our uniform dynamic proxies are assessed through two studies. First, we implement future method invocations in a way balancing the transparency of the future-specific static analysis of Praktikakis et al. [52] and the safety of the run-time support of Welc et al. [70], without requiring any specific static analysis or program transformations. Second, we report on several application scenarios for proxies from a platform we implemented for peer-to-peer programming [19], including classic scenarios such as structural conformance as well as more specific ones such as lazily evaluated predicates.

- We discuss our uniform proxies/uniformly virtual object model in the light of the proxification criteria introduced. In particular, we contrast our field access interception scheme with our own previous work [17], pointing out the lack of modularity of that initial approach. Then we further dissect transparency and investigate the pros and cons of its different faces in more depth. This allows us to
determine which application scenarios benefit most from proxies, and which are better implemented with specific language support or design patterns. Specifically, we point out that when transparency is a main concern, structural conformance is best implemented with more inherent language support, and proxies might not be worthwhile for implementing scenarios following wrapper and decorator design patterns, especially when considering the incurred overhead of reification and reflective execution of methods. Interestingly, these scenarios are among the most frequently mentioned cases for dynamic proxies.

Many authors have proposed performing code transformation or code generation to implement message interception or proxification. Several approaches have been proposed to add general behavioral reflection to Java (e.g., [26, 14, 72]). More related work can be found in the area of transparent distribution (e.g., [68, 46]); some of the afore-mentioned systems have been especially used towards that end. However, there are important differences in semantics and scope of the different approaches. Serving different purposes and starting from other design guidelines than our work, related systems perform similar but different transformations or fail to integrate with the existing proxies in Java, intercept field accesses, or deal with private or final features. As often, the devil is in the detail, and the lack of formal modeling of transformations is remarkable. We discuss through related work the pitfalls of vague semantics.

1.4. Roadmap

Section 2 dissects the original concept and implementation of dynamic proxies in Java. Section 3 discusses the limitations of dynamic proxies, and introduces criteria for proxification. Section 4 presents our approach to supporting dynamic proxies for classes, based on byte-code level program transformations. Section 5 presents a core calculus for Java with dynamic proxies, and extends it for uniform proxies. Section 6 presents various issues in the implementation of our transformations. Section 7 reflects on safety and transparency in future method invocations, and illustrates how uniform proxies can help provide both. Section 8 presents further application scenarios in a platform for peer-to-peer programming. Section 9 discusses transparency, security, modularity, and limitations. Section 10 overviews related work, including unsafe approaches to proxification in Java. Section 11 concludes with final remarks.

2. Background: Unfolding Java Proxies

This section dissects the concept of dynamic proxies introduced with Java 1.3.

2.1. Presentation and Terminology

Our appraisal includes a presentation of the types involved (see Figure 1) and perceived by programmers when manipulating dynamic proxies, and of the creation of dynamic proxy classes. For presentation simplicity, we henceforth may drop the qualifier “dynamic” when referring to proxies in the sense of Java reflection, unless confusion might otherwise arise. Furthermore, we omit package names when they are unambiguous, for example java.lang.reflect common to most types for reflection (except
public interface InvocationHandler {
    public Object invoke(Object proxy, Method meth, Object[] args)
        throws Throwable;
}

public class Proxy implements Serializable {
    protected InvocationHandler h;

    protected Proxy(InvocationHandler h) {
        this.h = h;
    }

    public static InvocationHandler getInvocationHandler(Object proxy)
        throws IllegalArgumentException {
    ...
    }

    public static Class getProxyClass(ClassLoader ldr, Class[] interfaces)
        throws IllegalArgumentException {
    ...
    }

    public static Object newProxyInstance(ClassLoader l, Class[] is, InvocationHandler h)
        throws IllegalArgumentException {
    ...
    }

    ...
}

Figure 1: Main types of the API for Java dynamic proxies: InvocationHandler and Proxy (excerpt).

...Class). We refer to such reflection classes as meta-classes, and to corresponding instances as meta-objects. Classes and interfaces presented in separate figures are reduced to those parts which are relevant in the present context, and exception handling may be omitted for brevity in code examples. A formalization of a subset of the proxification features currently supported in Java are presented in the next section.

2.2. Proxy Objects

A dynamic proxy is an object which conforms to a non-empty set of interfaces \(\{I_1, ..., I_n\}\) for which that proxy (‘s class) was created. The corresponding proxy class extends class Proxy [64] depicted in Figure 1, and implements all interfaces \(I_1, ..., I_n\). Conforming to all those types, a proxy can be cast to any of them, and hence any method defined in either of those interfaces can be invoked on the proxy.

2.2.1. Invocation handlers

Every proxy object has an associated object of type InvocationHandler (see Figure 1), which handles the method invocations performed on the proxy. More precisely, these invocations are reified and passed to the invocation handler through its invoke method as illustrated by Figure 2. The arguments for invoke include (1) the object on which the method was originally invoked (i.e., the proxy), (2) a reification of the method (an instance of class Method) that was invoked on the proxy, and (3) the actual arguments (as an array of Objects). The invoke method is hence capable of handling any method invocation, which manifests in that the type of its return value and its arguments are of the root Object type, and it is declared to throw Throwable.

\(^3\)The latter are different from meta-level objects which we will introduce in Section 3.
According to the specification [62], an exception of type ClassCastException is thrown if a value of a wrong type is returned by invoke. Any exception from invoke of a type not declared by the actually invoked method is wrapped in an exception of type UnknownThrowableException. This occurs also upon invocation of a method \( m \) on a proxy created for a set of interfaces \( \{ I_1, ..., I_n \} \) where at least two interfaces \( I_i \) and \( I_j, i \neq j \), declare method \( m \) with compatible return - and argument types, but with non-identical sets of exceptions. To ensure conformance, \( m \) will namely be declared to throw only exceptions declared by \( m \) in \( I_i \) and \( I_j \). A NullPointerException is thrown if \( \text{null} \) is returned by the invoke method but the return type of the method invoked on the proxy is a primitive type. In general, values of primitive types (e.g., \( \text{int} \)) need to be wrapped by corresponding objects (e.g., \( \text{Integer} \)).

The invoke method of invocation handlers can thus be seen as the antagonist of the invoke method implemented by meta-objects representing methods, i.e., instances of the Method class. While the latter method allows to defer to run-time the choice of which method to invoke, the former one provides a means of deferring to run-time the decision of what a given method performs, and is essential to the concept of proxy.

2.2.2. Proxy creation

The getProxyClass method in class Proxy expects as argument a set of interfaces defined by their respective Class meta-objects. When invoked, the method creates a proxy class, directly as byte-code, as a class implementing that set of interfaces – unless a proxy class has already been created for that set. Another argument to the method is a class loader to load the possibly created class.

In contrast to the getProxyClass method described above, newProxyInstance in addition instantiates the possibly generated proxy class. It hence takes an additional argument, which is an invocation handler, and returns an instance of that class which the specified invocation handler is associated with.

The Proxy class hence has a dual purpose. First, it serves as supertype for all proxy classes. Besides grouping functionalities common to all proxy classes, this makes it possible to easily verify through the instanceof operator whether a given object is a proxy. Second, the Proxy class contains class methods, described above, which permit the generation of proxies/proxy classes, thus serving as a factory [23].

Note that in certain cases it is impossible to create a proxy class for a set of interfaces [62]. Essentially, the procedure fails whenever conflicts would also arise if a
class was explicitly, i.e., statically, defined implementing the specified set of interfaces. Examples are the creation of a proxy class for two interfaces defining the same method with different unrelated return types, or for two non-public interfaces defined in two distinct packages. A proxy class for a set of interfaces of which all with package visibility are defined in the same package is created in that package, otherwise the package remains unspecified. In the following, we suppose for presentation the package in the latter case to be always the same.

2.3. Proxy Class Internals

Once these cases of incompatibility have been ruled out, the generation of a proxy class for a given non-empty set of interfaces \( \{ I_1, \ldots, I_n \} \) occurs as depicted in more detail in the following. A proxy class contains two constructors and a method body for each method of any of its interfaces \( I_i \) – including for those inherited from any of their respective super-interfaces. For presentation simplicity, we adopt in the following the name \( I_1 \ldots I_n \text{Proxy} \) for a proxy class created for a set of interfaces \( \{ I_1, \ldots, I_n \} \). According to the specification [62], the name space “$Proxy*” is reserved.

Consider for example a simple interface \( I \), defining a single method:

```java
import java.io.*;

public interface I {
    public String foo(Integer i) throws IOException;
}
```

Figure 3 outlines code of the proxy class generated for \( I \) in a schematic manner, chosen to accommodate our own implementation dealing also with the issues pointed out in the next section(s). Code fragments specific to custom interface \( I \) and its method(s) are italicized. Methods `equals`, `hashCode`, and `toString` inherited by every class from `Object` are handled just like custom methods, but are omitted from the Figure 3 for simplicity. They are also overridden by proxy classes, and invocations to them are forwarded to the invocation handler of the respective proxy. Other methods defined in `Object` are not overridden by proxy classes, as they are `final`.

We now present a simple example program which creates a proxy for type \( I \). In the example, the invocation handler associated with the proxy simply prints the name of any method that is invoked on the proxy to the standard output, subsequently forwarding the method invocation to another object \( \text{real}I \) implementing type \( I \). As mentioned, exception handling is omitted for brevity.

```java
final I realI = ...;
InvocationHandler h = new InvocationHandler() {
    public Object invoke(Object target, Method m, Object[] args)
        throws Throwable {
        System.out.println("Method " + m.getName() + " invoked");
        return m.invoke(realI, args);
    }
};
Class c = I.class;
I i = (I)Proxy.newProxyInstance(c.getClassLoader(), new Class[]{c}, h);
i.foo(10);
```

> Method foo invoked
import java.io.*;

public final class IProxy extends Proxy implements I {
    private IProxy() {}
    public IProxy(InvocationHandler h) { super(h); }

    public String foo(Integer i) throws IOException {
        try {
            Class c = I.class
            Method m = c.getDeclaredMethod("foo", new Class[]{Integer.class});
            return (String) h.invoke(this, m, new Object[]{i});
        } catch (IOException e) { throw e; }
        catch (Error err) { throw err; }
        catch (RuntimeException rex) { throw rex; }
        catch (Throwable t) { throw new UndeclaredThrowableException(t); }
    }
    ...
}

Figure 3: A schematic view of a sample proxy class created for the example interface I. Parts specific to I are italicized. The remaining parts are common across proxy classes and proxified methods.

3. Assessing Proxy Mechanisms

In this section, we elaborate on the limitations of proxies in general, and on dynamic proxies in Java in particular.

3.1. Proxy-inherent Limitations

The simplicity of the concept of proxy object accounts for its wide adoption as well as for its relative ease of implementation in object systems – in the simplest form following a design pattern. The same simplicity however limits the applicability of proxies in contexts such as behavioral reflection, where they have been used for associating a behavior in the form of a meta-level object with an existing base-level object. Here the proxy mimics a base object, but the two remain clearly distinguishable entities. While in certain scenarios this distinction is desirable, it is restrictive in the case of behavioral reflection, and has been one of the main driving forces for more inherent support for such purposes (cf. aspect-oriented programming [39]). For associating a behavior through a dynamic proxy as a meta-level object with an existing base-level object, the invocation handler associated with the corresponding proxy would most likely have to be given a reference to that object, and henceforth, the proxy would always have to be used instead of the original object, for instance also for self-invocations. However, when a method of the actual target object invokes a further method of that very object, there is no way of intercepting that invocation (“self problem” [41]). Similarly, self-references returned by a base-level object invoked through a proxy would have to be recognized as such (e.g., by the associated invocation handler in the case of Java’s proxies), such that again a proxy could be returned instead (“encapsulation problem” [41]).
The detecting and handling of all such situations is very hard. Base-level objects can for instance also return references to their fields which are difficult to identify as such, but would have to be shielded behind proxies as well.

3.2. Limitations of Java Proxies

In Java, proxies can currently only be created for interfaces. Hence, creating a proxy for every value returned by a base-level object requires return types of all methods of such an object to be interfaces, and recursively also the return types of the methods of those objects, etc. When striving for a uniform model of behavioral reflection with the current support for proxies in Java, the programmer basically would be constrained to program exclusively with variables of interface types, and make use of classes only in instantiations.

This limitation applies more generally to any application which would like to proxify objects of arbitrary application-defined types.

3.3. Limitations of the Extension Approach

Achieving uniform proxies, i.e., that any object can be proxified, is of course possible with inherent language support, or may be straightforward to achieve in a simple type system. The dynamically typed Smalltalk language makes it possible to easily create a proxy as an instance of a generic proxy class by simply overriding the default message handler doesNotUnderstand [54]. The extension approach for creating proxies chosen in Java, where a proxy for a given type is simply of a subtype of the former type, is also implementable as a pure library, i.e., without support from the runtime environment, yet with limited breadth. Since an instance of a proxy class must conform to the types it was created for, a proxy class created for a class $C$ would also have to subclass $C$ and override all members. We summarize in the following the obstacles encountered in the context of Java when straightforwardly applying this extension approach. Several of these have been pointed out previously by other authors, for example in the context of transparent distribution [65, 67]. Comparisons are further drawn in Section 10, which also relates alternatives such as the envelopment approach chosen by prominent implementations of behavioral reflection in Java. The goal is not to argue against any of the current features of Java, but simply to point to resulting complications when it comes to creating proxies through an extension approach. Other programming languages provide similar mechanisms and thus our observations are applicable beyond the Java language.

- An extension approach works well as long as everything is virtual, i.e., no declarations are sealed. In Java, not everything is virtual; proxies could for instance not be created for classes kept from being subclassed by being marked as final. Similarly, methods decorated by that same keyword could not be overridden.

- Members which are hidden might not be overridable. In Java, fields are hidden [42] ("shadowed"), and do not undergo dynamic dispatch. Though the semantics and rationale are different, private methods are also in some sense hidden, and thus can not be overridden by subclasses.
The coupling of initialization code between classes and super-classes, especially if not explicit, is another source of problems. In Java every constructor (except the no-argument constructor in the root class Object) must call a super-class constructor, whether this is explicitly coded as first instruction inside a given constructor, or a call to the default no-argument constructor of the super-class is automatically added [42]. Relying on such default constructors of super-classes for instantiation of proxy classes can yield side-effects (e.g., invoking arbitrary methods), and as a matter of fact, might also fail if such a constructor is hidden (i.e., private, see above).

Systematic programming with interfaces may help bypass these problems, and has been advocated for in different contexts (e.g., [60]). Current programs do however not abide to this principle and thus necessitate automatic transformation. This bears challenges of its own, some of which are similar to those inherently tied to Java’s type system [59, 30] identified in our context, e.g., how to deal with private methods.
Note that we focus on proxy objects, and thus do not address static methods or fields. Code transformations similar to the ones proposed later-on have been used in the past to proxify such features (see Section 10).

3.4. Proxification Criteria
Based on the previous observations, we define below criteria of “proxification” considered for the extensions proposed in the following and their subsequent evaluation.

**COMPLETENESS**: This twofold criterion quantifies the proportion of the types which can be proxified at all, and also how much of those types can be proxified. Typically, the impossibility of creating proxies for final classes in Java limits COMPLETENESS in the first sense, while the impossibility of overriding certain members can limit it in the second sense.

**SAFETY**: Proxifying a construct and giving the possibility to perform any actions instead of the ones defined by the original abstraction can lead to safety issues including unexpected exceptions and undesired side-effects.

**SECURITY**: Closely related to the issue of safety is that of SECURITY: using an entity instead of another one can obviously also lead to vulnerabilities.

**OVERHEAD**: Any proxification, or indirection used to achieve it, may introduce a performance overhead which must be considered.

**TRANSPARENCY**: This measure describes how aware the programmer is of the points at which proxification occurs. Behavioral reflection nicely demonstrates the impact of transparency of a proxification mechanism.

**MODULARITY**: The proxification of a given feature or any mechanisms used to achieve it should ideally be confined to that construct or its implementation and not interfere with other features or their proxification.
Often times tradeoffs have to be made, especially as several of these properties may be correlated in proxy implementations. The first criterion for instance describes the breadth of a proxification. Extending it beyond a certain point — even if possible — might however lead to unsafe or unsound behavior and thus be undesired. Similarly, transparency of a proxification mechanism is not actually desired in certain contexts as it may hamper safety. We will further dissect transparency later on in the context of different application scenarios.

4. Uniform Dynamic Proxies

This section proposes an enhancement of Java’s current implementation of dynamic proxies, aiming at increasing completeness by adding support for the creation of proxies for a set of types including a class type.

4.1. Uniformly Virtual Object Model

In essence, our approach builds on the principle applied for the generation of proxies for interfaces, that is, a proxy class for a set of types including a class is generated, when needed, at run-time as byte-code, loaded, and linked. In order for a proxy to be able to reify any action performed on it, its class must hence “override” all super-class members. To deal with the limitations outlined in the previous section, we propose in the following a set of nearly independent transformations, denoted by $T_{XY}$ for a specific transformation $XY$, performed at byte-code level upon class loading for dealing with those cases. Combined these yield a uniformly virtual object model, in which any instance member of a class can be overridden, and any class can be extended. Section 6 depicts the integration of this model into Java’s class loading procedure.

Note that the choice of an extension approach has been motivated by the desire of retaining as much as possible of the existing mechanisms and concepts of proxies for interfaces, but also by the fact that this approach adds no direct performance penalty in the common case of invocations of non-private non-final methods.

4.2. Proxy Types

As a direct consequence of the extension approach, a proxy class created for a set of types including a class $C$ must subclass $C$ and hence cannot subclass class Proxy. As elucidated in Section 2.2.2, it is however very useful to have a common supertype for all proxy types, be it for the mere purpose of testing whether an object is indeed a proxy. To that end, we introduce the ProxyType interface (see Figure 4).

The Proxy class still serves as super-class for all proxy classes created for a set of interfaces exclusively (see Figure 5), and hence implements ProxyType. This is depicted in Figure 6, which focuses on additions in the new backwards-compatible version of the Proxy class. Modifications from the original version are italicized.
public interface ProxyType extends Serializable {}

public interface AccessHandler {
    public Object get(Object proxy, Field field) throws RuntimeException;
    public void set(Object proxy, Field field, Object val)
        throws RuntimeException;
}

public class UnexpectedRuntimeException extends RuntimeException {...}

Figure 4: New auxiliary types to support proxies for classes. AccessHandler complements InvocationHandler to intercept field accesses.

Figure 5: Overview of proxy types and their relationships.

4.3. Access Handlers

Furthermore, we introduce a type AccessHandler to reflect the possibility of performing instance field accesses in addition to instance method invocations in the case of proxies for classes. Field accesses made on a proxy created for a set of types including a class are handled namely through methods get and set of an instance of that AccessHandler type associated with the proxy. The AccessHandler type hence complements the InvocationHandler interface. Such an instance must be passed to any proxy created for a class.

Class Proxy has been added variants of the class methods getProxyClass and newProxyInstance, thus enabling the lookup/generation of a proxy class for a class and (possibly empty) set of interfaces, including instantiation of that proxy class in the case of the latter method.

Exceptions can be thrown, similarly to the original Proxy class, whenever the signatures of methods of supertypes for which the proxy class is to be created conflict, or visibility problems occur as discussed in Section 2.2.2.

4.4. Field Accesses

Field accesses have to trigger method invocations in order to be proxified. With an extension approach, a solution to this — detailed below and formalized later in Section 5 — consists in transforming field accesses to invocations of getter/setter methods which are automatically generated for classes. While the use of such code transformations is not new, the importance lies in the exact transformation. Since in Java, like in many other languages, fields can not be overridden by subclasses, but hidden (see
public class Proxy implements ProxyType {
    protected InvocationHandler ih;
    protected Proxy(InvocationHandler h) {
        this.h = h;
    }
...

    public static AccessHandler getAccessHandler(Object proxy)
            throws IllegalArgumentException {...}
    public static Class getProxyClass(ClassLoader l, Class c, Class[] is)
            throws IllegalArgumentException {...}
    public static Object newProxyInstance(ClassLoader ldr, Class cl, Class[] is,
            InvocationHandler ih, AccessHandler ah)
            throws IllegalArgumentException {...}
}

Figure 6: Summary of augmented Proxy class. It is backwards compatible with the original class, i.e., all original features are retained. A program compiling and running with the original class remains valid.

Section 3.3), field access interception can not be achieved simply by defining a getter/setter method pair a-la `get f / set f` in a class C for each field f defined by C. Care must be taken that the information about the class C in which a field is declared is not lost. A solution to this consists in conveying information about the static type C through which a field is accessed in its respective getter/setter methods. To that end, we determine the name of such methods according to two respective functions R, W: F × C → M, where F, C, and M denote the sets of field, class, and method names respectively. We illustrate this below through three recursive subclasses assuming for this example R(f, C) = `get$C$f` and W(f, C) = `set$C$f`:

Original classes: \[\iff \mapsto \text{Transformed classes} \]

Observe the corresponding transformations in code accessing the fields of the above classes (source code for readability). The original code (left) and the code resulting from replacing those lines with corresponding transformations (right) in the original code have the same effect:
This scheme ensures that always the right variable is accessed. The specific corresponding transformation on Java programs is dubbed $T_{\text{fa}}$. In Section 9.4 we discuss an alternative, initial, design [17] whose implementation provides reduced MODULARITY.

Of course accesses to a field made in the access methods of that field are not transformed to ensure that the fields can actually be accessed and as well as to avoid endless recursion. Similarly, field accesses in field initializations coupled with field declarations are retained to ensure correct initialization.

Note that for simplicity the package name of a class is supposed to be part of the class name. The names of getter/setter methods for a field declared in a given class namely contain the name of the class (in addition to that of the field), but also that of the package containing the class (with occurrences of ’.’ being replaced by ’$’). Without this information, conflicts could occur when a class called $C$ in package $p_2$ subclasses another class $C$ in package $p_1$, and both classes declare a field of same name.

4.5. Visibility and Initialization

The above transformation $T_{\text{fa}}$ deals with the most important new aspect of class proxification – the interception of field accesses. For a fully practical solution, several intricacies of Java’s type system must however be considered.

4.5.1. Private fields and methods

The above transformations use a field’s visibility modifier for the access methods generated for the field, which does not enable the reification of accesses to fields which are declared as private. This stems from the fact that the dispatch of private methods, and thus corresponding getter/setter methods, does not start at the class of the invoked object, but rather at the class declaring the method – making use of the invokespecial rather than invokevirtual byte-code operator [42]. Circumventing this caveat is non-trivial; in the Spring framework [58] typically the problem is avoided by limiting interception in the AOP features to non-private methods. In the present context, getter/setter methods for private fields are defined with package visibility, the weakest visibility enabling overriding/dynamic dispatch. This is in itself however not sufficient, as it can lead to accidental overriding in proxy classes just as in the case of fields.
The similarity between the lookup for private methods and the lookup for fields suggests the adoption of a scheme for interception of application-defined private methods inspired by the one applied for field accesses, consisting in complementing private methods with stub methods, through which former methods are invoked. A stub method differs from the original method in its visibility qualifier (package visibility) and name. Akin to getter/setter methods, stub methods convey information about the method name and their class name. More precisely, stub method names are determined by a function \( U : M \times C \rightarrow M \) (e.g., \( U(m, C) = C$\#m)\). This avoids accidental overriding in subclasses, since, as described above, a class can very well declare a same private method as its super-class. Private methods are complemented by stub methods rather than modified directly because the renaming of methods declared to be native would invalidate lookup tables of corresponding native libraries. The transformations corresponding to private methods and private fields are dubbed \( T_{PM} \) and \( T_{PF} \) respectively. Note that \( T_{PF} \) only makes sense if \( T_{FA} \) is enabled.

4.5.2. Final classes and methods

The solution to circumventing the limitations introduced by the final keyword consists in handling final classes and methods as non-final ones internally, yet keeping track of these occurrences for the verification of loaded classes. The implementation of this model in the linking procedure is described shortly in Section 6.1. The transformation schemes for dealing with final classes and final methods are referred to as \( T_{FC} \) and \( T_{FM} \) respectively. As a consequence of \( T_{FM} \) — unlike in the original implementation of dynamic proxies — also methods tagged in the root object type Object as final can now be overridden by proxies to intercept corresponding invocations, even by proxies created for interfaces only.

4.5.3. Super-class constructors

The troubles with default constructors have been mentioned already in Section 3.3. As a countermeasure, we add a specific constructor to each class, which is parameterized by a class-specific initializer class. The name of such classes is abstracted by a function just like the names for getter and setter methods. In this case, the function has only one argument, as a constructor name is always the same as the class name: \( C : C \rightarrow C \). The thus named class is a public class containing itself nothing but a public no-argument constructor. In transformation \( T_{SC} \) every class is added a constructor with a single argument of that type, passed to the corresponding super-class constructor.

4.6. Uniform Proxy Class Internals

From the programmer’s perspective our scheme for proxifying classes integrates seamlessly with the original one in Section 2.2.2. If no class \( C \) is specified upon proxy creation, the two schemes behave identically. A proxy class for a class \( C \) is defined as subclass of that class \( C \) (and subtype of ProxyType), and overrides all original instance methods of its super-class(es). Proxy methods for (non-private) methods are still created in proxy classes exactly as described in Section 2.3.

To reify accesses to fields and invocations to private methods, a proxy class generated for a class also overrides all getter/setter and stub methods generated by the
transformations for its ancestor(s). Upon invocation of a getter or setter method, the corresponding field is reified and passed to the AccessHandler associated with the proxy by calling the get or set method respectively.

Regarding the package in which a proxy class is created, the rule given in Section 2.2.2 applies without modifications. That is, a proxy class can only be created for a set of types including a class of which some types have package visibility if they are indeed defined in the same package. Furthermore, proxy creation can fail in similar situations as with proxies for interfaces only (i.e., clashes in method declarations). In the case of a proxy class created for a class $C$ and an interface $I$, both defining the same method yet with different visibilities (i.e., anything except public in the case of $C$), the proxy class implements that method as public.

Suppose a class $C$ implementing interface $I$ from Section 2.3, adding a field $bar$:

```java
public class C implements I {
    public String bar;
    public String foo(Integer i) throws IOException {...}
}
```

The following lines illustrate the creation of a proxy for class $C$. The invocation handler $h$ from Section 2.3 is reused here:

```java
final C realC = new C();
AccessHandler ah = new AccessHandler() {
    public Object get(Object target, Field f) {
        System.out.println("Field "+f.getName()+" read");
        return f.get(realC);
    }
    public void set(Object target, Field f, Object val) {
        System.out.println("Field "+f.getName()+" written");
        f.set(realC, val);
    }
};
Class cl = C.class;
C c = (C)Proxy.newProxyInstance(cl..., new Class[]{cl}, null, ah, h);
c.bar = "hello";
```

The proxy class created upon the invocation of the newProxyInstance method defined in the augmented Proxy class yields the pseudo code depicted in Figure 7 (with $C(Proxy) = ProxyInit$). The figure also shows why field accesses in field initializations are not transformed to calls to access methods: with the super-class constructor being called at the beginning of the constructor and due to initialization semantics in Java the access handler is installed only after such fields would be initialized and thus after calls would be made to it.

5. A Core Java Language for Proxies

We now introduce Proxifiable Java (PJ), a simple core language for Java with support for proxies, in order to argue for safety of our extensions and compare proxification schemes. (Readers focusing on practical descriptions of our approach may choose to directly move to the next section.) For presentation clarity we sometimes abbreviate notation, e.g. `impl` stands for `implements`, `Obj` abbreviates `Object`.  

> Field bar written
import java.io.*;

public final class CIPrxy extends C implements ProxyType, I {
    private AccessHandler ah;
    private InvocationHandler ih;

    private CIPrxy() {}
    public CIPrxy(AccessHandler ah, InvocationHandler ih)
    {
        super(new ProxyInit());
        this.ah = ah;
        this.ih = ih;
    }

    public AccessHandler getAccessHandler()
    { return ah; }
    public InvocationHandler getInvocationHandler()
    { return ih; }

    public String foo(Integer i) throws IOException {...

    public String get$C$bar()
    {
        try {
            Field field = C.class.getDeclaredField("bar");
            return (String)ah.get(this, field);
        }
        catch (RuntimeException r) {
            throw new UnexpectedRuntimeExeption(r);
        }
    }

    public void set$C$bar(String bar)
    {
        try {
            Field field = C.class.getDeclaredField("bar");
            ah.set(this, field, bar);
        }
        catch (RuntimeException r) {
            throw new UnexpectedRuntimeExeption(r);
        }
    }

    ...
}

Figure 7: Extract of sample proxy class for proxifying instances of class C in addition to instances of interface I. Parts specific to C are italicized. Specific parts of I are like in Figure 3.

interface S ::= int l EXT T [M]
class L ::= class C EXT C IMPL T [F, F, K, M, B]
constructor K ::= C (T, F, F, this, J) method M ::= T m (T, T)
body B ::= (l, l)
value v ::= l(C) | l(T) | new Str(s)
term t ::= v | x | t.f(C) | t.f(C) = t | t.m(T) | new C(i) | (T) t | new T[l, i] | t[n]

Figure 8: Syntax of Proxifiable Java (PJ). The main syntactical additions with respect to FJ are interfaces I, locations l(C), arrays (l(T)) represent corresponding array locations), as well as Method meta-objects and character strings (String).

5.1. Syntax

PJ is inspired by core calculi for Java such as Classic Java [22], or, in particular, Featherweight Java (FJ) [33], a widely employed core imperative pass-by-value
calculus for Java. PJ supports overriding, but other features such as exceptions and overloading or modifiers (e.g., final, private) are omitted for simplicity.

5.1.1. Basics

Figure 8 presents the syntax of PJ, which we will slightly extend later-on. \( \tau \) represents a sequence of elements \( z_{1}\ldots z_{q} \), with \( q \) denoting the maximum index by default. A program is a set of class declarations \( L \) and interface declarations \( S \). Every method body consists in a sequence of terms \( t \), with the last one being implicitly returned after evaluation. A class declaration \( L \) encompasses a set of field declarations \( T \overline{j} \), a constructor \( K \), and a set of method declarations with corresponding bodies \( M \overline{B} \). A constructor defines formal arguments corresponding in order to all the fields of the class, including inherited ones. Interfaces only define method headers \( M \overline{B} \). In the absence of super-interfaces we sometimes omit the extends clause. PJ thus extends FJ with interfaces \( I \), and further adds references by introducing locations \( l(C) \) (for brevity we sometimes just write \( l \) omitting the type \( C \) when it is not used) as values to which constructor invocations are reduced. Furthermore, field hiding is introduced to faithfully capture field manipulations. For dynamic invocation handling, PJ also introduces a simple meta-class Method, as well as character strings and arrays. The latter two are needed, respectively, to capture the names and arguments of reified methods. Strings are of the form \texttt{new Str(s)} and handled as values like all objects in FJ, meaning that they are not reduced to locations. Arrays are created by terms of the form \texttt{new T[]{...}} and reduced to special array locations \( ll(T[]) \). The \( n \)-th element of a term representing an array is accessed by indexing: \( t[n] \). Here, \( t \) can be itself an array term, leading to multi-dimensional arrays (with corresponding access \( t[n] \)). PJ does not support assignments into arrays (e.g., \( t[n]=t' \)), as these are not necessary for our purposes. Their omission, conversely, allows us to simplify the calculus, while still faithfully representing the considered subset of Java’s array features. The restriction namely helps us avoid the unsafe part of covariant subtyping in Java arrays.\(^4\) \( T \) represents all types, including array types, and is introduced to simplify rules.

Method encapsulates some of the functionalities of class Class in Java for brevity; that is, a Method meta-object can be obtained directly for a given type name and method name, without having to obtain a Class meta-object first. Besides special classes such as Object, Method, and String, we also introduce the Proxy class. It abstracts the creation of a proxy by representing it as a constructor call on that class, parameterized also here by the names of the desired interfaces directly rather than class meta-objects. A created object is an instance of the respectively instantiated proxy class. All class names of the form \_Proxy are reserved for proxy classes.

5.1.2. Field hiding

Later-on, we extend PJ in two steps for encoding the most invasive transformation necessary to achieve proxies for classes. The first step is necessitated as the FJ model does not inherently model effects of type checking. More precisely, without

\(^4\)E.g. String[] s = \texttt{new String[]{"1", ...}; Object[] o = s; o[0] = \texttt{new Integer(4)};
field hiding (and other Java features) in FJ type checking is of a purely assertive nature. Yet, compilation of real Java programs includes checks for determining the static type for a given field access, yielding an output which persists in the generated bytecode. PJ only provides “location-specific” terms for field reads and writes – \( t.f(C) \) and \( t.f(C)=t' \) respectively. These make the static type of the term through which the field is accessed explicit. In a first extension to PJ described shortly, we will add traditional location-unspecific field access terms and describe an automatic transformation to PJ. It is important to note that the (seemingly location-unspecific) field assignments in constructors retained from FJ are not really terms. They are never executed, i.e. reduced, in FJ or PJ.

5.2. Auxiliary Functions

PJ assumes that all classes as well as interfaces are preloaded into a table \( DT \) (declaration table\(^5\)) This includes all possible proxy classes, i.e., without conflicting interfaces. Figure 9 presents auxiliary functions for tracking method types (\( mtype() \)), method bodies and formal arguments (\( mbody() \)) for implementing overriding, and fields of given classes (\( fields() \)). As expected the first auxiliary method is the only one to be defined for interfaces. If an auxiliary function is not explicitly defined for a given set of parameters, \( \bullet \) is assumed. The fields of a class \( C \) in PJ are represented as a record \( fields(C) = [T_f(C')] \) with each field \( f \) tagged by its type \( T \) as well as by its declaring class \( C' \) to allow for hiding of fields by subclasses. (In contrast, the location \( T \) specified in a field access such as \( t.f(T) \) represents the static type of the term \( t \) used for the access.) Since a class can thus contain several fields of a same name, the auxiliary function \( fproj() \) allows to project the fields of a given class onto a specific name (see rules [FIELD-PROJ-INCL] and [FIELD-PROJ-EXCL]). Method, String, and Proxy are special classes whose declarations are “hardwired” due to their reflective nature, just like the root object class \( Object \) is implicitly defined (rule [OBJ-CLASS]). The \( InvocationHandler \) interface is pre-loaded into the table ([INVHANDLER-INFI]).

Figure 9 further presents three predicates. Predicate \( moverride() \) ensures that if a class \( C \) defines a method \( m \) then it overrides it with the correct signature \( T \rightarrow T' \). \( munique() \) verifies the absence of conflicting definitions for a method \( m \) in a set of interfaces. It is used by the predicate \( implements() \) which ensures that a class \( C \) implements all methods of a set of interfaces. \( munique() \) is also used by rule [PROXIES], which implicitly defines the shape of possible proxy classes; \( munique() \) is used here to ensure that no two interfaces implemented by a same proxy class have conflicting definitions of a same method.

5.3. Reduction

Dynamic semantics of PJ are presented as contextual operational semantics in Figure 10. Evaluation contexts \( E \) are terms with a hole in them representing the current term under reduction. The reduction relation is of the form \( \langle t, \mathcal{L} \rangle \rightarrow \langle t', \mathcal{L}' \rangle \) where \( t \) is the current term being reduced and \( \mathcal{L} \) is the object store. \( \mathcal{L} \) contains records of

\(^5\)FJ uses a class table \( CT \) due to the absence of interfaces.
the form \( f(C) : v \), where \( f_i \) is a field declared by class \( C_i \) and \( v_i \) is its value. In
the case of an array location \( l_i \), \( L(l_i) = \langle T \rangle \). We could easily represent all store entries
in a uniform way, e.g., by assuming them to be of the form \( f_i^0 \langle T \rangle \) with \( f_i^0 \) being the
same for all \( i \) in the case of arrays, or by introducing separate stores for objects and for
arrays respectively. We believe the present solution is more intuitive.

Object constructor invocations are reduced to new locations by rule [LOC-R], unless in the case of Method and Proxy which are handled by [METH-CONS-R] and [PROXY-R] respectively. In the former case, care is taken that the designated method actually exists. In the latter case, the absence of conflicting methods in the designated interfaces is attested. This is achieved indirectly, by verifying whether a proxy class for that set of interfaces exists in the table. As mentioned, Strings are handled by value and corresponding constructor terms are thus not reduced to locations. Array constructor terms are reduced to corresponding special locations (rule [ARR-R]).

Method invocations are evaluated by substituting the reduced actual arguments for the formal arguments ($\bar{\theta} / \bar{x}$) and the invoke location for this in the term sequence representing the respective method body. Method bodies are reduced to the last value once every term in the sequence has been reduced (rule [RET-R]). Class Method only
understands the \texttt{invoke} method whose semantics are defined by rule \texttt{METH-INV-R}; such a dynamic invocation is reduced to an actual invocation, provided the actual arguments match the formals. Rules \texttt{FIELD-ACC-R} and \texttt{FIELD-ASS-R} represent field reads and writes respectively, taking into account hiding. Rule \texttt{ARR-ACC-R} defines the indexing of arrays. The rule ensures that the index $n$ is within bounds (elements in the store are indexed internally starting from 1). Casting of an object (rule \texttt{OBJ-CAST-R}) or array (rule \texttt{ARR-CAST-R}) succeeds and yields the cast entity, provided that it is cast to a super-type of the original type.

5.4. Typing and Soundness

Figure 11 presents subtyping rules ($T \lesssim T'$) and type checking rules ($\Gamma \vdash t : T$). New rules with respect to FJ are introduced notably for dealing with interfaces, proxy creation terms, and field accesses.

5.4.1. Basics

The subtype relation $\lesssim$ is reflexive (rule \texttt{REFL-S}) and transitive (rule \texttt{TRANS-S}). Rules for sub-classing (\texttt{extends} on classes – \texttt{CL-DECL-S}), interface inheritance (\texttt{extends} on interfaces – \texttt{INTF-DECL-S}) and interface implementation (\texttt{implements} – \texttt{IMPL-DECL-S}) are intuitive. Array subtyping (rule \texttt{ARR-S}) follows subtyping of the types of the elements, but as mentioned, does not lead to the well-known array covariance issues due to the absence of assignments into arrays. All singleton types are subtypes of \texttt{Object}, including meta classes such as \texttt{Proxy} and \texttt{Method}. In addition, as in Java, PJ allows an array of an arbitrary type to be typed as \texttt{Object} (rule \texttt{OBJ-S}). Otherwise, no application-defined method $m$ could have an argument $x_i$ of array type because it would be impossible to represent such an argument when reifying all arguments $\pi$ of $m$ by an array of type \texttt{Object[]} as required by \texttt{Method} or \texttt{InvocationHandler}.

A typing environment $\Gamma$ is a set of variable-to-type bindings of the form $x:T$ (rule \texttt{ENV-T}). Rule \texttt{METH-T} types method call terms in a quite standard manner by assigning them the corresponding return type, provided that the actual arguments match the formal ones. Constructor calls are typed by the respectively designated class (rule \texttt{CONS-T}) or array type (rule \texttt{ARR-CONS-T}) if the arguments can be typed accordingly. Since the \texttt{Proxy} class is defined implicitly, corresponding constructor calls are handled separately by rule \texttt{PROXY-CONS-T}. Such terms are typed as \texttt{Object} to align with the return type of the \texttt{newProxyInstance} method in class \texttt{Proxy} of Java, the method which the \texttt{Proxy} constructor in PJ mimics. String also requires an individual rule \texttt{STR-CONS-T}. In contrast, the constructor of the special class \texttt{Method} can be typed with the default \texttt{CONS-T} rule. Rules \texttt{U-CAST} and \texttt{D-CAST} deal, respectively, with up- and downcasts of objects or arrays alike. The former casts are always safe, while the latter may be possible and thus will be resolved at reduction. Rule \texttt{S-CAST} deals with “stupid” casts where the target type is unrelated to the subject type. The Java compiler rejects expressions containing stupid cast, but these are necessary to formulate type soundness as a subject reduction theorem (cf. [33]).
Theorem 2 (PJ Progress).

As a consequence of our syntax and typing rules, programmers can use casts in order to access fields in super-classes in PJ the same way as in Java. Suppose two classes \( C \) and \( C' \) such that \( C'' \preceq C \) and \( C'' \neq C \). \( C \) and \( C' \) both declare each a field \( f \) of a type \( T \). A syntax like \((C)\_t.f(C)\) can be used to access the field \( f \) of the super-class \( C \) on a term \( t \) which is typed as \( C'' \) (e.g., \( t \) is a formal method argument \( x \), or recursively a field \( f' \), of declared type \( C' \)). If \( t \) reduces to a location \( l(C'') \) with \( C'' \preceq C' \), the value stored for the field \( f \) of class \( C \) will indeed be returned rather than that of \( C' \). However, it is important to note that, conversely, typed field reads and writes are not substitutes for casts. More precisely, it is not sufficient to write \( t.f(C) \), with \( t \) a term of expected type \( C' \), to force the read of field \( f \) in class \( C \neq C' \). The term \( t \) must be preceded by a cast \((C)\_t\) for any \( t \) of expected type \( C' \neq C \). Rule [FIELD-ACC-T] could have been defined in a more lax manner to support such a shortcut, but this would have been a departure from our goal of modeling Java’s field hiding faithfully.

Figure 12 presents typing for interfaces (rule [INTF-OK]), methods (rule [METH-OK]), and classes (rule [CLASS-OK]). As mentioned, all classes (as well as interfaces) are preloaded into the declaration table \( DT \), including all possible proxy classes. Typing of an interface ensures the absence of conflicts in method declarations within its parent interfaces, and within them. Method typing ensures that a method implemented by a class does not have any conflicting definitions in any interfaces implemented by the class and, if implemented by the super-class, is overridden correctly. Finally, class typing uses the method typing rule to ensure that any method implemented by a given class \( C \) is conflict-free, and further ensures that any method defined by an interface implemented by \( C \) is indeed fully implemented by \( C \) (possibly by inheritance from \( C \)’s super-class). Class typing reflects field hiding, and also prohibits subclassing of proxy classes by the application. This is enforced in PJ as it is in line with the current implementation of dynamic proxies in Java (where generated proxy classes are final) as well as our extended implementation, and also for simplicity as we will see shortly when formulating proxies for classes.

5.4.3. Soundness of PJ

5.4.2. Typing field accesses and classes

As a consequence of our syntax and typing rules, programmers can use casts in order to access fields in super-classes in PJ the same way as in Java. Suppose two classes \( C \) and \( C' \) such that \( C'' \preceq C \) and \( C'' \neq C \). \( C \) and \( C' \) both declare each a field \( f \) of a type \( T \). A syntax like \((C)\_t.f(C)\) can be used to access the field \( f \) of the super-class \( C \) on a term \( t \) which is typed as \( C'' \) (e.g., \( t \) is a formal method argument \( x \), or recursively a field \( f' \), of declared type \( C' \)). If \( t \) reduces to a location \( l(C'') \) with \( C'' \preceq C' \), the value stored for the field \( f \) of class \( C \) will indeed be returned rather than that of \( C' \). However, it is important to note that, conversely, typed field reads and writes are not substitutes for casts. More precisely, it is not sufficient to write \( t.f(C) \), with \( t \) a term of expected type \( C' \), to force the read of field \( f \) in class \( C \neq C' \). The term \( t \) must be preceded by a cast \((C)\_t\) for any \( t \) of expected type \( C' \neq C \). Rule [FIELD-ACC-T] could have been defined in a more lax manner to support such a shortcut, but this would have been a departure from our goal of modeling Java’s field hiding faithfully.

Figure 12 presents typing for interfaces (rule [INTF-OK]), methods (rule [METH-OK]), and classes (rule [CLASS-OK]). As mentioned, all classes (as well as interfaces) are preloaded into the declaration table \( DT \), including all possible proxy classes. Typing of an interface ensures the absence of conflicts in method declarations within its parent interfaces, and within them. Method typing ensures that a method implemented by a class does not have any conflicting definitions in any interfaces implemented by the class and, if implemented by the super-class, is overridden correctly. Finally, class typing uses the method typing rule to ensure that any method implemented by a given class \( C \) is conflict-free, and further ensures that any method defined by an interface implemented by \( C \) is indeed fully implemented by \( C \) (possibly by inheritance from \( C \)’s super-class). Class typing reflects field hiding, and also prohibits subclassing of proxy classes by the application. This is enforced in PJ as it is in line with the current implementation of dynamic proxies in Java (where generated proxy classes are final) as well as our extended implementation, and also for simplicity as we will see shortly when formulating proxies for classes.

5.4.3. Soundness of PJ

Theorem 1 (PJ Subject Reduction). \( \Gamma \vdash t : T \rightarrow t' \Rightarrow \exists T' \leq T | \Gamma \vdash t' : T' \).

Straightforward proof by induction on derivation of \( \rightarrow \).

Theorem 2 (PJ Progress). Suppose \( t \) is a closed, well-typed normal form. Then either of the following holds:

(a) \( t \) is a value
(b) \( \exists E \mid t = E[\Gamma[v]] \land (v = \mathbf{new}T'(\_\_)) \lor v = l(T') \lor v = ll(T')) \land T' \neq T \)
(c) \( \exists E \mid t = E[\Gamma[v]] \land L(\ll) = [v_1, \ldots, v_n] \land q \leq n \)
(d) \( \exists E \mid t = E[\mathbf{new} \mathbf{Proxy}(\ll(\mathbf{str}), v)] \land L(\ll) = [\mathbf{new} \mathbf{str}(s)] \land \)
   (d.1) \( \exists s_i \in D \mathbf{str}^{\Gamma}(s_i) \neq \mathbf{intf} I_\_ \)
   (d.2) \( \exists s_i \in \mathbf{str}, mtype(m, s_i) \neq \mathbf{\neg munique}(m, s) \)
(e) \( \exists E \mid t = E[\mathbf{new} \mathbf{meth} [\mathbf{new} \mathbf{str}(s), \mathbf{new} \mathbf{str}(s')] \] \land mtype(s, s') = \mathbf{\bullet} \)
Figure 11: Subtyping and type checking in PJ.

Figure 12: Class and interface typing in PJ.

(f) $\exists E \mid t = E[(\text{str}, \text{inv}(v, T))] \land L(l) = [\text{elem} : \text{mth} \text{Str}(m), \text{cl}(\text{mth}) : \text{new Str}(T)] \land L(l) = [v_1, \ldots, v_q] \land \text{mtype}(m, T) = T_1 \times \ldots \times T_q \rightarrow T'$ \land
\begin{itemize}
  \item[(f.1)] \( v \neq l'(C) \)
  \item[(f.2)] \( v = l'(C) \land \)
  \item[(f.2.1)] \( C \not\subseteq T' \)
  \item[(f.2.2)] \( q \neq q' \)
  \item[(f.2.3)] \( \exists v_i \in \pi \mid (v_i = l''(T') \lor v_i = \text{new } T' \lor v_i = \text{ll } T') \land T' \not\subseteq T_i. \)
\end{itemize}

Proof by induction on typing derivations. The main additions with respect to FJ are cases (c)-(f): (c) is a consequence of the introduction of arrays and reflects the case of an array index which is out of bound; (d) reflects the case where a proxy is created for a set of interfaces including non-existing ones (d.1) or with conflicting method definitions (d.2); (e) represents an attempt to create a Method object for a non-existing method. Finally, (f) represents the case where such a method object is used in an inappropriate way. (f.1) represents the case where it is used on a value which is not a singleton location, and (f.2.1) stands for the case where the location does not conform to the expected type. With (f.2.2), the wrong number of arguments is provided, and with (f.2.3) there is at least one actual argument which does not match its formal argument type. Case (b) is augmented with respect to FJ to capture failed casts on locations (for singleton objects and for arrays).

5.5. \PJ_{Lu}: Automating Lookup of Field Declaration Sites

Of course programmers do not want to be burdened with specifying the class \( C \) in which a given field \( f \) to be accessed is declared, in particular as the choice of \( C \) is deterministically given in PJ as it is in Java. We thus introduce \PJ_{Lu}, a simple extension of PJ which takes care of determining the declaring class \( C \) of an accessed field, and is automatically “compiled” to PJ. Figure 13 shows the relevant definitions.

5.5.1. From PJ to \PJ_{Lu}

The syntax of \PJ_{Lu} is simply defined by two additional, standard, location-unspecific terms for field reads \((t.f)\) and writes \((t.f=t)\). Two respective type checking rules are also provided for such terms (rules \([\text{LU-FIELD-ACC-T}]\) and \([\text{LU-FIELD-ASS-T}]\)). Additionally, Figure 13 defines a set of transformation rules \( T_{Lu} \) for translating well-typed \PJ_{Lu} programs into well-typed semantically equivalent PJ programs. Once it has been asserted that all classes in the table \( DT \) for \PJ_{Lu} are well-typed, the transformation is applied to the entire table \( DT \). This way of handling language extensions is in fact similar in spirit to the strategy of reducing Featherweight Generic Java to Featherweight Java by type erasure [33].

5.5.2. Soundness of \PJ_{Lu}

The following theorems mirror Theorems 1 and 2 respectively. They mirror those theorems after transformation \( T_{Lu}[\![DT]\!] \).

**Theorem 3 (PJ_{Lu} Subject Reduction).** Theorem 1 holds for \( T_{Lu}[\![DT]\!] \).
We proceed now to formalizing $\tau_{lu}$, the most invasive transformation mandated by proxies for classes, as an extension to $\tau_{lu}$ dubbed $\tau_{lu}$. Several of the foundations for this process were already laid with PJ and $\tau_{lu}$, most importantly the introduction of hidden fields. Figure 14 provides a high-level overview of the relationships between the different languages PJ$_lu$. 

5.6. Formalizing $\tau_{lu}$ in PJ

We proceed now to formalizing $\tau_{lu}$, the most invasive transformation mandated by proxies for classes, as an extension to $\tau_{lu}$ dubbed $\tau_{lu}$. Several of the foundations for this process were already laid with PJ and $\tau_{lu}$, most importantly the introduction of hidden fields. Figure 14 provides a high-level overview of the relationships between the different languages PJ$_lu$. 

\[ \begin{array}{c|c|c}
\hline
\text{term} & t::= v | x | t.f | t.f = t | t.f(C) | t.f(C) = t | t.m(T) | \text{new} C(T) | (T) t | \text{new} T(t) | T \\
\hline
\end{array} \]

\[ \begin{array}{c|c|c}
\hline
\Gamma \vdash t : T & \Gamma \vdash t : C & \Gamma \vdash t : T & T \leq T_q \\
\hline
fproj(f,C) = \lfloor \ldots, T_q f(C_q) \rfloor & \Gamma \vdash t : C & fproj(f,C) = \lfloor \ldots, T_q f(C_q) \rfloor & \Gamma \vdash t : C \\vdash t' : T_q \\\n\hline
\end{array} \]

\[ \begin{array}{c|c}
\hline
\text{[LU-FIELD-ACC-T]} & \text{[LU-FIELD-ASS-T]} \\
\hline
\end{array} \]

\[ \begin{array}{c}
T_{\text{lu}}[DT] = T_{\text{lu}}[DT(C)] \\
T_{\text{lu}}[\text{class } C_{\text{-class } M B}] = \text{class } C_{\text{-class } T_{\text{lu}}[M B]} \]_\text{this} \Gamma \\
T_{\text{lu}}[T m (T \pi)(T_1)] = T m (T \pi)(T_{\text{lu}}[T]^{\Gamma}, \pi) \]_\Gamma \\
T_{\text{lu}}[v] = v \\
T_{\text{lu}}[x] = x \\
T_{\text{lu}}[t.f(C)] = T_{\text{lu}}[v]^{\Gamma}, f(C) \]_\Gamma \\
T_{\text{lu}}[t.f(C)] = T_{\text{lu}}[v]^{\Gamma}, f(C) = T_{\text{lu}}[v]^{\Gamma} \]_\Gamma \\
T_{\text{lu}}[t.m(T)] = T_{\text{lu}}[v]^{\Gamma}, m(T_{\text{lu}}[v]^{\Gamma}) \]_\Gamma \\
T_{\text{lu}}[\text{new } C(T)] = \text{new } C(T_{\text{lu}}[v]^{\Gamma}) \]_\Gamma \\
T_{\text{lu}}[(T) t] = (T) T_{\text{lu}}[v]^{\Gamma} \]_\Gamma \\
T_{\text{lu}}[\text{new } T(t)] = \text{new } T(t) T_{\text{lu}}[v]^{\Gamma} \]_\Gamma \\
T_{\text{lu}}[t.m(T)] = T_{\text{lu}}[v]^{\Gamma}, m(T_{\text{lu}}[v]^{\Gamma}) \]_\Gamma \\
\hline
\end{array} \]

Proof by induction on derivation of $\rightarrow$. Two cases are added with respect to Theorem 1, corresponding to the two new kinds of terms introduced by PJ$_lu$. As $\tau_{lu}$ completes, any location-unspecific field reads or writes have been transformed to location-specific ones by the respective rules for $\tau_{lu}$. It is easy to see that the fields — as well as types determined for the individual subterms — in those rules are identical to the respective ones in the [LU-FIELD-ACC-T] rules.

**Theorem 4 (PJ$_lu$ Progress).** Theorem 2 holds for $T_{\text{lu}}[DT]$. 

Proof by induction on typing derivations. Through transformation $\tau_{lu}$, location-unspecific field access terms are transformed to location-specific ones. The only (seemingly) location-unspecific field access terms remaining are those in constructors, which, as mentioned already in Section 5.1.2, are not actually executed.
Figure 14: Overview of PJ. PJ₁₆ extends PJ syntactically, by adding location-unspecific field reads and writes. These are transformed by $T_{16}$. The core grammars underlying PJ₁₆ and PJ₁₄ are the same, yet through specific definitions and reduction rules PJ₁₆ supports Field meta-objects, and an alternative instantiation of the Proxy class in order to support proxies for classes. The transformation $T_{16}$ works together with $T_{14}$ to translate the PJ₁₄ portion of PJ₁₆.

5.6.1. Syntax and auxiliary functions

The main required additions in terms of syntax and definitions are depicted in Figure 15. In essence, the syntax underlying PJ₁₄ is retained. X, which is presented as part of the syntax in Figure 8 but is mostly introduced to simplify and generalize the rules for static and dynamic semantics across presented variants of PJ, is augmented with a new meta-class Field introduced for the reification of fields, analogously to Method for reifying methods. The class Field contains two methods for dealing with the activation of reified field reads and writes respectively whose implementations are hardwired through two respective reduction rules [FIELD-CLASS] in Figure 15. We introduce an AccessHandler interface whose definition is preloaded into DT just like Invocationhandler (rule [ACC-HANDLER-INTF]).

To avoid introducing void into the calculus, the set method of the AccessHandler type returns an instance of Object, as opposed to the full-fledged interface presented in Figure 4. This is aligned with the definition of assignments in PJ, where a field assignment simultaneously triggers an update to the object store and reduces to the assigned term (see rule [FIELD-ASS-R] in Figure 10). Rule [CL-PROXIES] captures the shape of proxies for classes. In addition to the constituents in rule [PROXIES] in Figure 9, a proxy class for a class contains an additional field ah representing the access handler, as well as methods $M^{TC^{\epsilon}}$ for overriding field access methods.

5.6.2. Dynamic and static semantics

A noteworthy trait of the methods for triggering field reads and writes in class Field is that they reduce to field access method invocations, rather than directly to field reads and writes. The corresponding reduction rules are [FIELD-META-ACC-R] and [FIELD-META-ASS-R] respectively in Figure 16. Direct field accesses would lead to bypassing proxification in reified field accesses. The consequence of reducing invocations of reified fields to invocations of corresponding getter and setter methods is the possibility for endless recursion if an access handler simply evaluates a resolved Field object on the proxy through which the access was intercepted. Observe that the same can
5.6.3. Formalizing \( T_{\alpha} \)

Figure 17 presents the transformations \( T_{\alpha} \) necessary to translate all field reads and writes to invocations of generated getter and setter methods respectively. These methods are generated as described by the generation function \( G_{\alpha} \), and are added as part of the transformation \( T_{\alpha} \) performed on any class \( C \) contained in table \( DT \). It is important to note that the only field read and write terms \( t_f(C) \) and \( t_w(C) \) to appear in \( \Pi_{\alpha} \) programs after transformation \( T_{\alpha} \) are those \textit{inside} the field getter/setter methods generated by \( G_{\alpha} \). \( T_{\alpha} \) is not applied to those terms, as shown in Figure 17.

This observation allows us to summarize the proxification of field reads/writes in \( \Pi_{\alpha} \) and to tie some open knots. As mentioned above, field read or write terms for a field declared in a class \( C \) appear only in getter/setter methods generated for \( C \), which are overridden by any corresponding proxy class \( C_{Proxy} \) such as to reify respective

\[
\begin{align*}
\text{meta } X & := \text{Mth|Fld} \\
\text{mtype}(get, Fld) &= \text{Obj} \rightarrow \text{Obj} \\
\text{mtype}(set, Fld) &= \text{Obj} \times \text{Obj} \rightarrow \text{Obj}
\end{align*}
\]

Rule [FIELD-CONS-R] describes the creation of field reifications. Rule [CL-PROXY-R] is required to allow the Proxy class in \( \Pi_{\alpha} \) to be available with two different constructors, abstracting the two newProxyInstance methods from the augmented Proxy class of Figure 6. Rule [CL-PROXY-CONS-T] describes the typing judgement corresponding to this second use of new Proxy (...) terms in \( \Pi_{\alpha} \).
reads/writes (see rule \([\text{CL-PROXIES}]\) in Figure 15). These, in turn, yield invocations to the generated getter/setter methods when evaluated on an object conforming to \(C\). When
evaluated on an instance of \( C \), the generated getter/setter methods then indeed access the field, while if evaluated on an instance of \( C \cdot \text{Proxy} \) we enter recursion the same way as described for reified method invocations earlier (the reified field access will trigger the field access method which recursively reifies the access and passes it to the \text{AccessHandler}). This is an important observation, as the values of inherited fields in the case of a proxy class \( C = C' \cdot \text{Proxy} \) are undefined (explicitly set to \( \bullet \)) according to rule \([\text{CL-Proxy-R}]\) in Figure 16, and getter/setter methods of the super-class \( C' \) are overridden thus bypassing corresponding field access terms. Lastly, by separating generation and transformation that way, we get the desired feature that the accesses to inherent fields of proxy classes, namely \( \text{ih} \) and also \( \text{ah} \) in the case of proxies for classes, are not reified, but are effectuated directly (through the generated getter/setter methods). Proxification of these fields could be achieved by allowing proxy classes to be subclassed. As mentioned, we have chosen to not allow application classes to do so (see rule \([\text{CLASS-OK}]\) in Figure 12) mostly to be in line with Java’s semantics. In addition, when supporting proxies for classes (rule \([\text{CL-Proxies}]\)) such proxy class subclassing would lead to an infinite number of recursive proxy classes.

5.6.4. Soundness of \( \text{PJ}_n \)

Now we can proceed to stating preservation and progress guarantees on \( \text{PJ}_n \). The following theorems hold after \( T_n[\ll DT\rr] \) completes.

**Theorem 5 (\( \text{PJ}_n \), Subject Reduction).** Theorem 1 holds for \( T_n[\ll DT\rr] \).

Proof by induction on derivation of \( \rightarrow \). The only significant new terms added with respect to \( \text{PJ}_n \) are due to the introduction of the Field meta-class. The relevant definitions \([\text{Field-\ldots-Class}]\), \([\text{Field-Proxies}]\), and \([\text{Meta-Class}]\) (Figure 9) define the permissible creation and uses of such objects; these definitions match the reduction rules \([\text{Field-Cons-R}]\), \([\text{Field-Meta-Acc-R}]\), and \([\text{Field-Meta-Ass-R}]\) and none other.

**Theorem 6 (\( \text{PJ}_n \), Progress).** Suppose (after application of \( T_n[\ll DT\rr] \)) \( t \) is a closed, well-typed normal form. Then either of the following holds:

(a) \( t \) is a value
(b) \( \exists E \mid t = E[(T\cdot v) \setminus v = \text{new } T'(\ldots) \lor v = \text{ll}(T') \land T' \not\subseteq T} \)
(c) \( \exists E \mid t = E[I[n]\cdot \setminus \mathcal{L}(ll) = [v_1,\ldots,v_q] \land q \leq n} \)
(d) \( \exists E \mid t = E[\text{new } \text{Proxy}(ll(str),v)] \setminus \mathcal{L}(ll) = [\text{new } \text{Str}(s)] \setminus \)
   (d.1) \( \exists s_i \in \pi \cdot DT(s_i) \neq \text{inf } I_{\ldots} \)
   (d.2) \( \exists s_i \in \pi \cdot \text{mtype}(m,s_i) \neq \bullet \lor \neg \text{munique}(m,\pi) \)
(e) \( \exists E \mid t = E[\text{new } \text{Proxy}(\text{new } \text{Str}(s_1),\text{ll}(str),\ldots)] \setminus \mathcal{L}(ll) = [\text{new } \text{Str}(s_2),\ldots,\text{new } \text{Str}(s_q)] \setminus \)
   (e.1) \( DT(s_1) \neq \text{class } C \ldots \)
   (e.2) \( DT(s_1) = \text{class } \ldots \text{Proxy } \ldots \)
   (e.3) \( \exists i \in [2.q] \cdot DT(s_i) \neq \text{inf } I_{\ldots} \)
   (e.4) \( \exists s_i \in \pi \cdot \text{mtype}(m,s_i) \neq \bullet \lor \neg \text{munique}(m,\pi) \)
(f) \( \exists E \mid t = E[\text{new } \text{mth}(\text{new } \text{Str}(s),\text{new } \text{Str}(s'))] \setminus \text{mtype}(s,s') = \bullet \)

30
Proof by induction on typing derivations. One new case is added with respect to Theorem 4 — besides the last three cases which are added to deal with dynamic field accesses — namely the new, inserted, case (e) for creation of proxies for classes. It represents the cases permitted by the $[C\text{-PROXY-CONS-T}]$ typing rule but failing the premises of $[C\text{-PROXY-R}]$. (e.1) represents the case where the class doesn’t exist, and (e.2) when the class is a proxy class itself. The other two cases are analogous to the creation of proxies for interfaces. Thanks to the reduction of field accesses through corresponding meta-objects to invocations of corresponding generated methods (rule $[\text{FIELD-META-...-R}]$) — including in proxy classes for classes — and to the passing of respective calls to access handlers, no uninitialized inherited fields are ever accessed in a proxy class. The last three cases, as mentioned, deal with erroneous uses of class Field. (h) represents the attempt to obtain a meta-object for a non-existing field. In (i) a field read is performed on a value of an inappropriate kind (i.1) or which does not conform to the expected type (i.2). Finally, in (j) a field write is similarly either performed on a value of inappropriate kind (j.1) or type (j.2.1), or with a term of inappropriate type (j.2.2).

6. Implementation Issues

This section discusses implementation choices and consequences of our extensions.
6.1. Applying Transformations

As mentioned previously, the transformations described in the previous section are performed at load time. When a class \( C \) is to be loaded, it goes through the following components (see Figure 18):

Analyzer: Class \( C \)’s byte-code is analyzed, and the extension and visibility constraints of \( C \) conflicting with subclassing/overriding, e.g., occurrences of \textbf{final} or \textbf{private}, are identified.

Verifier: The class \( C \) is verified, in particular against the constraints of previously loaded classes.

Virtualizer: The transformations described in the previous section are applied. Beyond this point \( C \) is uniformly virtual.

Linker: The class \( C \) is linked, and is ready to be used.

Though possible, this loading procedure has not been implemented as a user class loader, as the loading of only some classes in a way bypassing that loader (by using another user class loader, cf. Figure 18) would have strongly hampered SAFETY and SECURITY [27]. While one could imagine only applying a subset of the transformations, it is imperative that the same transformations are applied to all classes upon their loading. As illustrated by \( T_{\text{sa}} \), transformations namely may incur changes on classes as well as classes using the former ones. Along the lines of several other extensions to Java (e.g. [1]), we thus make use of a preprocessor integrated with the virtual machine’s (default) class loading and linking path. More precisely, the analyzer, verifier, and virtualizer are all parts of an instrumented system class loader. As illustrated by Figure 18, classes can still be loaded with user class loaders before being passed on to that system loader.

![Figure 18: Class loading and delegation. User class loaders 1 and 2 are simply examples and do not refer to particular applications. Here the former loader delegates to the latter. Class \( \text{C} \) can be loaded by either loader. The Virtualizer is a specific component introduced by our approach. The Verifier keeps track of visibility constraints of previously verified and subsequently virtualized classes.](image-url)
6.2. Increasing Safety

As mentioned previously, the parameterization of the functions \( R(f, C) \) and \( W(f, C) \) for naming field access methods, as well as of \( U(m, C) \) for naming stub methods after the class \( C \) in which the respective members are originally defined, avoids accidental overriding in subclasses thus supporting SAFETY. The parameterization of constructors by \( C() \) serves a similar purpose.

The sample functions given so far, e.g. \( m \circ C \) for a stub for method \( m \) in class \( C \), have only been chosen for illustration. In fact, these functions also involve a secret key to hash \( m \) and \( C \) and are sensitive to execution parameters such as time to avoid generating predictable outcomes.

This measure not only further decreases the chances of accidental overriding of field access methods, stub methods, or constructors, but also avoids intentional overriding. Otherwise, by observing one such name, e.g., through the printout of a stack trace upon the occurrence of an exception, specifically designed subclasses could be introduced easily into the system, compromising SAFETY (see Section 3.4), and possibly SECURITY. Issues related to the latter concern, and in particular how to avoid that knowledge on generated methods acquired at run-time can be exploited, are discussed in Section 9.2.

6.3. Resolving Meta-Objects

As already pointed out, the illustrations provided for the creation of dynamic proxies in the case of a set of types consisting of only interfaces, but also in the case of a set including a class, are schematic. According to Figures 3 and 7 namely, all relevant meta-objects (representing methods and fields respectively) are resolved (i.e., looked up) at every method invocation or field access targeting an instance of a proxy class. In practice, this repeated resolution is replaced by a static, lazy, one. More precisely, proxy class instances retain Method and Field objects once looked up in an internal table which is defined as static and thus shared by all instances of such a class.

Note at this point that the Method object looked up for a given method invoked on a proxy does not necessarily represent the exact method which would have been executed had the object not been a proxy. This is due to the fact that invocations of (non-private) methods are dynamically dispatched, and that the type of the variable through which the invocation is performed is “lost”. In particular, a proxy being called through a method \( m \) declared in two distinct interfaces \( I_1 \) and \( I_2 \) both implemented by the proxy class, can not distinguish whether it has been called through \( I_1 \) or \( I_2 \). Just like with the original implementation of dynamic proxies in Java, the method meta-object resolved upon that invocation is the one corresponding to the first of the interfaces \( I_1 \) and \( I_2 \) to have appeared in the array of types passed upon creation of the proxy (proxy class). Classes take precedence over interfaces in the case of proxies for classes.

More useful information is however obtained upon field accesses with our extension. Since the information of the declaring class of an accessed field is contained in the name of the used getter or setter method, the resolved meta-object in the case of a field access reification can reflect faithfully the class containing the accessed field (which is however not necessarily the static type of the variable through which the field was accessed). PJ\(_A\) faithfully reflects the semantics described above for our extension.
6.4. Overhead

The transformations performed on code to support proxies for classes nicely illustrate the tradeoff between completeness and overhead. Removing the effect of the final keyword may affect performance by reducing the number of opportunities where the just in time (JIT) compiler can perform method inlining. However, JIT compilers such as Sun’s HotSpot™ virtual machine can also inline methods which are not final, and as long as no subclass overriding such an inlined method is loaded (e.g., a proxy class), the JIT compiler does not have to recompile an affected class. This occurs in our case only when a proxy class is loaded, and hence no sensible overhead was measured due to TFC and TFM in performance measurements. The main overhead associated with each of those transformations, and the only one in the case of TSC, becomes then the cost of performing the transformation itself, which occurs only once for a loaded class and is negligible for the applications investigated.

The transformations for field accesses (TFA) and private members (TPM and TPF) are however more expensive in terms of overhead. Furthermore, it turns out that the latter category has a stronger impact than the former one. This is proof of a good programming discipline, as it reflects the level of encapsulation that is achieved with respect to fields – if few or no members were private TPM and TPF would not add overhead. In any case however, the overheads are not as drastic as one might expect. These observations are conveyed by Figure 19, which elucidates results of performance measurements (normalized) obtained with different sets of transformations enabled. These results were computed with the SpecJVM benchmark suite and Jikes [2] on an Intel core 2 Duo 2.66GHz, running Redhat Linux 4. (Though enabled, TSC is not mentioned in the figure, for the reason described above).

7. Study 1: Transparent and Safe Futures

This section illustrates the benefits of uniform proxies for Java by implementing future invocations in a way balancing transparency and safety.
7.1. Explicit Futures in Java

When implementing futures [73] in Java, programmers are currently required to identify upfront which methods/subprograms might be called asynchronously, and are forced to make use of the type parameterized Future and Callable interfaces defined in package java.util.concurrent outlined in Figure 20. In short, type Future\(<T>\) is used on the caller side when the respective logical return type of an (asynchronous) invocation would have been \(T\), and the called object will in fact implement Callable\(<T>\). A programmer can achieve the asynchronous execution of a callable object by instantiating the predefined FutureTask with that object. Executing the future task (invoking run) then leads to invoking the call method on the callable object. Any exception raised by the call is delivered to the caller at the point where it invokes the get method, wrapped up in an ExecutionException; otherwise the actual result is returned.

```java
package java.util.concurrent;

public interface Future\(<T>\) {
    \(T\) get() throws InterruptedException, ExecutionException;
    ...
}

public interface Callable\(<T>\) {
    \(T\) call() throws Exception;
}

public class FutureTask\(<T>\) implements Future\(<T>\), Runnable {
    public FutureTask\(\) Callable\(<T>\) clble) throws NullPointerException {...}
    public void run() {...}
    ...
}
```

Figure 20: Interface Future and related types as defined in the standard Java API for futures.

7.2. Transparency

Pratikakis et al. [52] propose static analysis to make futures less explicit. More precisely, the authors strive for transparency, which can be split into three aspects:

TYPE: The (return) type of a future call appears to be the return type \(T\) of the method actually executed, and not Future\(<T>\) (or Callable\(<T>\) on the callee side).

IDENTITY: A future object, as placeholder for the value to be computed eventually, and the value then actually computed, appear as the same logical entity (cf. Section 3.1).

ASYNCHRONY: The “lazy nature” of a future object is masked. It can be passed around like any other object, e.g., as argument to a method call, though the underlying asynchronous call computing that very object might not have completed yet. Any call on such a future object is then blocked until the computation followed up.
A specific library call `Async.invoke(o.m(...))` is used to indicate future invocations in programs (here method `m` on object `o`) and guide the subsequent analysis which tracks potential occurrences of future objects, wrapping them in code to achieve the above transparency. Thus, a fourth form of transparency for future invocations is not a declared goal:

**CALL:** An invocation yielding a future object appears the same way as a regular (synchronous) invocation.

Such transparency is considered to be relevant in other settings for futures or analogous alternative scenarios for proxies. The above properties can all be generalized by defining them over proxies instead of futures.

### 7.3. Safety

Welc et al. [70] observe that **asynchrony transparency**, without supportive run-time mechanisms, may hamper consistency. The concurrent execution of a future invocation and its **continuation** — the code between the future invocation and the actual access to the return value — can namely lead to a different observable behavior than the sequential execution of instructions the way they appear in the code. A flagrant example is given by an exception raised during a future call only after several actions have already been performed as part of the continuation.

The approach of Welc et al. [70] leverages optimistic transactional mechanisms developed earlier [69]. A future call and its continuation are handled like two concurrent transactions, whose potentially conflicting actions (i.e., actions on shared data) must respect **serializability** [31]. If violations are observed, both the future call and its continuation are rolled back. Welc et al. [70] describe a class `SafeFutures` offering the same interface as `FutureTask`, but ensuring that the observable behavior of a forked future invocation executed in parallel with its continuation are the same as that of their sequential execution.

### 7.4. Safe Futures with Proxies

Based on this class `SafeFutures` and uniform proxies, we devise in the following an implementation of futures which combines the **type transparency** of the approach of Pratikakis et al. [52] with the **asynchrony transparency** of the approach of Welc et al. [70], and adds **call transparency**, without requiring the specific program transformations of the former work. Figure 21 sketches the implementation. Class `BackToTheFuture` makes it possible to create a proxy for an arbitrary object, through which *any* method of that object can be (indirectly) invoked asynchronously, transparently, and safely. Assume an instance of class `C` introduced in Section 2.3; the future invocation and the reference to the future object are *italicized*:

```java
C c = new C();
C cFut = BackToTheFuture.futurify<C>(c);
String s = cFut.foo(); /* future call */
... /* asynchronous continuation */
System.out.println(s); /* synchronization point */
```

---

6Rollback capabilities can be achieved in other less intrusive ways, cf. [40].
Calling `futurify` leads to creating a dynamic proxy with an instance of `AsyncHandler` for handling both method invocations and field accesses. While an action of the latter kind is directly relayed to the proxified object, a method call triggers the instantiation of class `SafeFuture` with a `SafeCall` (a subtype of `Callable`), which will actually perform the call. A proxy representing the future object is returned, which is handled by an instance of `FutureHandler`. Any call on that proxy will then block the caller on the `SafeFuture` until the underlying future call has completed.

```java
public class BackToTheFuture {
    public static <T> T futurify(T t) {
        AsyncHandler h = new AsyncHandler(t);
        Class c = t.getClass();
        return (T) Proxy.newProxyInstance(..., c, null, ..., h, h);
    }
}

class AsyncHandler implements InvocationHandler, AccessHandler {
    private Object orig;
    public AsyncHandler(Object orig) { this.orig = orig; }
    public Object invoke(...) { Method m, Object[] args... { ... start transaction */
        FutureHandler h = new FutureHandler(future);
        return Proxy.newProxyInstance(..., m.getReturnType(), null, h, h);
    }
}

class SafeCall extends Callable {
    private Method m;
    private Object on;
    private Object[] args;
    public SafeCall(Method m, Object on, ...) { ... }
    public Object call() { return m.invoke(on, args); }
}

class FutureHandler implements InvocationHandler, AccessHandler {
    private SafeFuture fut;
    public FutureHandler(SafeFuture fut) { this.fut = fut; }
    public Object invoke(...) { Method m, Object[] args... { ... /* end transaction */
    }
}
```

Figure 21: Balancing safe and transparent futures with proxies for classes. An instance of `AsyncHandler` handles all future invocations performed on a given object after “futurify-ing” it. Class `FutureHandler` allows for handling of field accesses and method invocations on future objects; the evaluation of the generating future invocation is forced upon such a nested action.
8. Study 2: A Manifold Experience with Uniform Proxies

This section describes several application scenarios of proxies within a same research platform for peer-to-peer (P2P) programming. The scenarios cover typical ones such as structural conformance, but also more specific ones such as lazy remote argument passing, or the expression of remotely evaluated predicates. These scenarios allow us to further analyze the tradeoffs observed when putting uniform proxies to work and thus the limitations of dynamic proxies.

8.1. Borrowers and Lenders

These mechanisms have been implemented within a same abstraction called borrow/lend (BL) [19]. The BL abstraction is essentially a higher-level representation of a distributed hashtable (DHT) [55, 53, 61], bearing resemblances with the tuple space [24] abstraction, and simplifying P2P programming with respect to existing rather low-level specific APIs such as JXTA [28].

With the BL abstraction, peers interact indirectly through shared resources, acting in two roles. These roles are represented by respective classes Borrower and Lender, instantiated by peers whenever they intend to borrow or lend resources, respectively. Figure 22 provides an overview of an application programmed with BL, and Figure 23 presents the core interfaces and classes. Instances of the two classes can be viewed as communication endpoints of components [51]. They are instantiated with a type parameter which represents the first of three criteria by which a resource borrower can describe the resources it is interested in:

Type. Borrower interests are expressed for objects of a given type $T_b$, with which the type $T_l$ of a resource borrowed from a lender must conform. Different “depths” $d$ of type conformance between $T_l$ and $T_b$ are possible: $T_l \preceq T_b$. A depth of $d = 0$ represents nominal conformance, i.e., $T_l$ has to be a declared subtype of $T_b$ or $T_b$ itself. This reflects the regular, nominal, subtyping in Java (noted $\preceq$ in PJ). A depth $d \geq 1$ represents structural conformance: with a depth of $d = 1$, $T_l$ only has to provide all the members declared by $T_b$; with a depth of $d = 2$ types of fields and method arguments in $T_l$ have to conform at depth $d = 1$ only to those of $T_b$, etc.

Key. Lenders can explicitly attach a key in the form of an array of bytes to a resource when lending that resource. This key plays the role of access control mechanism, and a corresponding argument is hence present on the borrower side as well.

Predicates. Queries can involve content filters expressed as predicates based on the members of the resource type. These will be illustrated through an example shortly.

8.2. Resources

The resource objects through which interaction between borrowers and lenders takes place (see Figure 24) can exhibit different semantics, depending on the application at hand. ValueResources for instance are passed by value between nodes, and ReferenceResources by reference.
As argued in earlier work [19], resources in P2P applications are however not always either "small", location-independent objects (e.g., events – ideal for pass-by-value semantics) or "large", location-dependent ones (e.g., services – typical for pass-by-reference semantics). They can have any intermediate sizes, and in certain cases one might want to download a copy of a given resource while in other cases a remote interaction with (a copy of) the resource will suffice. In addition, an asynchronous decentralized query mechanism such as achieved through the BL abstraction can yield several replies of which not all will be needed. Corresponding resources whose instances are initially passed by reference and can be subsequently copied by value subtype LazyResource. They bridge the gap between the corner cases of resources invariably passed by value (ValueResources) and such passed by reference (ReferenceResources), thus allowing the BL abstraction to seamlessly integrate (1) publish/subscribe interaction [50, 18] through the former type of resources and (2) a
lookup service for remote objects [6] through the latter type of resources.\textsuperscript{7}

Two variants of lazy pass-by-value semantics are [19]:

1. \textit{Explicit} lazy pass-by-value arguments are invoked remotely until they are explicitly downloaded by a borrower. This lazy argument passing style has also been used as motivating example.

2. The transfer of \textit{implicit} a lazy pass-by-value argument starts at its first invocation.

In the former case, the transfer is triggered through the specific \texttt{download} method. This method is implemented in a decorator pattern style. That is, a resource passed to a borrower through its \texttt{Inbox} is in fact a proxy, which encapsulates specific logic for implementing the \texttt{download} method. In the latter case, the invocation of any resource-specific method is intercepted by the proxy, initiating the transfer. Implementations of the two can be combined [29]. A third variant of lazy argument passing is [43]:

3. \textit{Imperative} lazy arguments are transferred immediately by value upon remote invocation of a method, but the body of the method can start to execute before they have been entirely received.

These last semantics are very similar to futures, but apply to the invokee side. They target methods whose arguments include larger ones and are able of performing some computation (e.g., starting transactions, setting up connections) before the transfer of these larger arguments has completed.

8.3. \textit{Contract Methods}

The concept introduced by the \texttt{download} method has been systematically expanded to \textit{contract methods} capturing individual characteristics of the different resource types. These are predefined methods for individual resource types, which reflect the consequences of the use of such resources. While some of these methods can be implemented such as to do nothing (corresponding proxies will provide the implementations), others can be specifically implemented by a resource class to override default behavior. Annotations can be used for further clarification.

Through the \texttt{setDownload} contract method for instance, a borrower can choose between the two lazy semantics variants (explicit is default) for a given resource object. Another example is \texttt{setInvocationStyle}, which allows a borrower to switch back and forth between asynchronous (future-style) and synchronous invocations for a given resource. Depending on the chosen invocation style, an actual call to \texttt{download} (like an implicitly triggered download of such a resource) returns immediately (future semantics), or blocks until the resource has been transferred (disabled). The transfer protocol to be used can be explicitly specified with the second \texttt{download} method.

\textsuperscript{7}The BL abstraction encompasses many further abstract resource subtypes, such as \textit{dynamic resources} (devoid of statically defined interfaces), \textit{replicated resources} (fault tolerance through \textit{consistent} replication, cf. [7]), or \textit{replaceable resources} (automatic updating of resources passed by value). Different communication protocols have been explored in the context of BL, e.g., [20, 21, 5].
public interface Resource {
    public Object deref() throws NoSupportException;
    public boolean isLocal() throws NoSupportException;
    public void setProtocol(Protocol p) throws NoSupportException;
    public void setQoS(QoS qos) throws NoSupportException;
}

public interface ValueResource extends Resource, Serializable {}

public interface ReferenceResource extends Resource, Remote {
    public void setInvocation(boolean asynch, RemoteExceptionHandler h)
        throws NullPointerException;
}

public interface LazyResource extends ValueResource, RemoteResource{
    public void setDownload(boolean automatic);
    public void download() throws RemoteException;
    public void download(Protocol p) throws RemoteException,
        NoSupportException;
}

public interface RemoteExceptionHandler {
    public void handle(RemoteException re);
}

Figure 24: Basic resource types in the BL abstraction.

8.4. Illustration: Exchanging Music

Consider the (in)famous scenario of songs shared throughout the Internet as objects [32, 8]. Figure 25 proposes a Java type Song for incarnating such objects. A Song instance conveys information about (1) the title, (2) the artist, and (3) the actual track (an object of type javax.sound.sampled.AudioInputStream). A music track can now be shared among peers as follows (exception handling omitted for simplicity):

Song lSong = new Song("Next love song", "Next boy band", ...);
Lender<Song> sLender = new Lender<Song>(lSong, ...);
Song lSong = sLender.setConstraints();
lSong.setProtocol(new FTP(...));
sLender.activate();

The call (italicized) to setProtocol is performed on a proxy returned by setConstraints, used not only for contract methods but also for “recording” predicates representing content filters. This is illustrated by the following example in which a peer expresses interest in specific songs:

Borrower<Song> songs =
    new Borrower<Song>(new Inbox<Song>() {
        public void deliver(Song bSong) {
            if (!Jukebox.isQueued(bSong));
            bSong.download(new FTP(...));
            Jukebox.queue(bSong);
        }
    }, ...);
Song pSong = songs.setConstraints();

41
pSong.setInvocation(true, new RemoteExceptionHandler{});

pSong.setConformance(1);

pSong.getArtist().equals("Next boy band");

songs.activate();

Duplicates are filtered by storing songs in a jukebox and checking whether a freshly received song is already present. A jukebox (see Figure 25) can be queried for a song represented by bSong at Line 4 through isQueued. This method simply queries the underlying queue through contains. Internally, this leads to calling the equals method on elements. Figure 25 shows the implementation of this method for songs. It consists simply in comparing elements based on their titles and artists through the getTitle and getArtist methods respectively. The corresponding calls performed on the new song are remote; thus, a local copy is first created in isQueued to avoid repeating these calls.

Only if not already present, the song referred to by bSong is then downloaded by invoking the download method at Line 5. Thanks to the lazy synchronization of that procedure, achieved by making the call (italicized) on a proxy, the song can be enqueued by the jukebox before having actually been entirely received at that point – releasing the corresponding thread. Similarly, all invocations made in the context of predicates for the actual borrower criteria are made on proxies (Lines 10-12). At Line 11, interests are expressed in objects which are instances of Song or of any type T providing the same public members as Song (T ⪯ Song) and which are of a certain band, i.e., their getArtist method returns a given name “Next boy band”.8 None of the proxified calls on Songs would be possible with the current non-uniform proxies in Java, simply because Song is a class.

9. Discussion

This section discusses oimplications of our uniform proxies and limitations.

9.1. Transparency

We first assess transparency with respect to our two studies and then analyze our observations. Table 1 summarizes our findings.

9.1.1. Future method invocations

We believe that the explicit “futurization” of an object with our futures presented in Section 7, alike the one by Pratikakis et al. [52], is not a drawback. It reminds the programmer of the restrictions that still apply, such as with asynchronous exceptions. Take the case of a try..catch block around a future call, intended at handling exceptions potentially caused by that call. The end of such a clause should bound the call’s continuation – otherwise an exception might be thrown “too late”. In the worst case, the future object is a return value for the enclosing method. This case is rarely detailed by implementations of futures which strive for ASYNCHRONY TRANSPARENCY. An

8A disjunction of conditions is achieved by expressing the conjunctions on different proxies.
public class Song implements LazyResource {
    public String getTitle() throws RemoteException {...}
    public String getArtist() throws RemoteException {...}
    public AudioInputStream getStream() throws RemoteException {...}
    public Song(String title, String artist, AudioInputStream s) {...}
    public boolean equals(Object o) {
        try {
            Song s = (Song)o;
            return (s.getTitle().equals(getTitle()) &&
                    (s.getArtist().equals(getArtist())));
        } catch(Exception ex) {
            return false;
        }
    }
}

public class Jukebox extends Thread {
    private static Queue<Song> queue = new ConcurrentLinkedQueue<Song>();

    public static boolean isQueued(Song s) {
        return queue.contains(new Song(s.getTitle(), s.getArtist(), null));
    }
    public static void queue(Song s) { queue.add(s); }
...
of predicates, proxies which are used to express borrower criteria do not attempt to mimic specific objects obtained in response to a corresponding query.

(a) Structural conformance is representative of a wrapper pattern. ASYNCHRONY TRANSPARENCY does not apply to structural conformance. Quite conversely, TYPE TRANSPARENCY can be seen as part of the actual goal of structural conformance, and can be definitely achieved in the sense that a type does not need to be added specific features such that a structurally conforming proxy can be generated (this is not entirely achieved in BL as a resource object must implement Resource). CALL TRANSPARENCY in this context refers to the obtention of a proxy to access a given object; ideally one would be able to assign an object of a given type to a variable of a structurally conforming type with implicit creation of a proxy. In other terms, CALL TRANSPARENCY would be desirable in this case, but is not achievable with proxies.

(b) Implicit lazy pass-by-value arguments fare just like (implicit) future invocations as they could be implemented without introducing specific root types for corresponding objects (as LazyResource in BL), except that this leads to undesirable (in this case) CALL TRANSPARENCY, as corresponding objects are obtained implicitly as method arguments. Inversely, explicit lazy arguments do not achieve CALL TRANSPARENCY (the download must be triggered explicitly) but in return hamper desired TYPE TRANSPARENCY, as the only way to indicate that an object is a priori remote but can be conveyed by value upon demand is through a root type. ASYNCHRONY TRANSPARENCY is however retained. Imperative lazy arguments fare exactly like future objects due to their strong similarity and are thus omitted from the table.

(c) Predicates achieve desirable TYPE TRANSPARENCY and avoid undesirable CALL TRANSPARENCY to a certain level. That is, as illustrated in the example, a program text allows to infer that a proxy obtained through setConstraints is a proxy, but with nested calls on the proxy it becomes more difficult, especially if a proxy is assigned to a variable before further constraints are expressed. ASYNCHRONY TRANSPARENCY is very desired (the query will be evaluated in the future, repeatedly) and is well achieved in this case.

(d) Contract methods represent a decorator pattern scenario. ASYNCHRONY TRANSPARENCY does not apply to such contract methods. CALL TRANSPARENCY is desirable here, as the presence of additional functionality is expressed through the type of the object and is thus omnipresent. As a downside, TYPE TRANSPARENCY is not achieved, as a resource class must implement the Resource interface even if the methods of that interface are implemented by the proxies. This does not imply changes in signatures of logical methods, but does impose a super-type with own methods in the present case. In a proxy-based decorator pattern in general, adding a method to an object requires the creation of a sub-type of the corresponding type for the method to be statically invocable, which forces the use of interfaces or at least non-final classes.
9.1.3. Analysis

Overall, proxies represent a good means to implement future invocations. This is conveyed by Table 1, which presents an overview of the forms of transparency achieved by the different application scenarios for proxies presented in Sections 7 and 8. ASYNCHRONY TRANSPARENCY applies to (and is well achieved by) mechanisms striving for some form of flow decoupling (futures, lazy arguments), or even decoupling in time (subsuming flow decoupling) such as predicates (which also provide decoupling in space). Those seem to benefit most from support for proxies in general and thus from our extensions presented earlier to broaden the scope of dynamic proxies.

In contrast, and quite interestingly, the least benefits of proxies in terms of transparency are garnered by structural conformance (type decoupling – a typical motivation for a wrapper pattern) and for contract methods (implementation decoupling – a typical motivation for a decorator pattern). It seems thus that one might be better off implementing wrappers and decorators explicitly as design patterns rather than through proxies, and that if transparency is a main concern, a language approach (e.g. [9, 25]) is probably best suited.

Obviously the set of application scenarios for proxies is potentially unlimited. In many other rather common scenarios such as object replication or persistence, transparency may desirable more broadly.

9.2. Security

As a sign of the times, the Java programming language has undergone nearly unprecedented intensive investigations in terms of SECURITY. An in-depth presentation of corresponding issues raised by general-purpose reflective extensions to Java, which is beyond the current context, can be found in [13].

9.2.1. Proxies

Core notions in Java security are protection domains, permissions, and policies [27]. Protection domains correspond to certificates for signing classes, and/or URLs for obtaining classes, and have an associated set of permissions. (If a class maliciously exploits the permissions associated with its protection domain, the principals associated with that domain can be held responsible.) Java system classes are part of a system protection domain which by default includes all permissions. A security policy in Java governs the permissions granted to the different protection domains.

Two key concepts underlying security in Java are (1) the principle of least privilege and (2) the concept of permission intersection. The former principle states that a piece of code should operate with the smallest possible set of privileges. The latter concept requires that, when performing a piece of code, the entire set of protection domains represented by classes on the execution stack at that point include the permissions necessary for executing that code.

In the context of dynamic proxies, handlers are the central players with respect to SECURITY. The (augmented) Proxy class, as well as created proxy classes are namely, just like any system classes, given all permissions. When an object accesses another object via a proxy, the only relevant classes added to the execution stack are thus handler classes. Besides a class implementing the InvocationHandler interface, this
potentially includes a class implementing `AccessHandler` in the case of proxies for classes. On the one hand, care must be thus taken when inserting a proxy between a caller and a callee, to not make the interaction impossible by associating an instance of a handler class with an insufficient set of permissions with that proxy. On the other hand, one can exploit this to dynamically introduce security barriers. Rather than providing an untrusted party with direct access to (instances of) a class, that party can be urged to access (instances of) that class through dynamic proxies. The corresponding handlers can then at run-time decide on granting permissions or not.

9.2.2. *Introspection*

The deflection of reified field accesses and method invocations to methods instead of in situ direct reduction of corresponding terms to allow for interception has illustrated the need to take a closer look at introspection classes.

Ensuring that introspection meta-objects (instances of classes such as `Field` or `Method`) representing the structures of linked classes do not reflect changes made at load-time improves *security* in addition to *safety*. A deeper integration of our uniform virtual object model in the Java runtime goes thus hand-in-hand with instrumented introspection classes such as `Class` to avoid exposing added fields and methods. Otherwise, the benefits of using secret hashing functions to generate stub or field access method names have limited benefits. If no `Method` objects are returned for such generated methods upon class inspection and the Verifier in the class loading procedure rejects e.g. classes with homonymous members, then insights gathered on names of specific generated methods for instance through stack traces output in the case of exceptions can not be used to infiltrate an ongoing execution. As mentioned in Section 6.2, the parameterization of hashing functions by execution parameters makes it hard to seek for vulnerabilities to exploit on a longer timeframe.

Note that the problem of protection of such generated or modified members is not specific to our work, but is encountered in various systems relying on code modifications. A general practical roadblock is that the introspection classes, such as all system classes, are protected by copyright terms and may not be distributed if modified.

9.3. *Completeness*

Dynamic proxies can currently not be created for primitive types. One might argue that this is not a limitation of dynamic proxies, but rather a consequence of Java’s hybrid type system which fuses objects with built-in primitive types. In addition, since primitive types do not have any members, there might be no need to intercept/override any calls to values of primitive types at all. The case of future invocations however illustrates that the lack of proxies for primitive types does reduce *completeness*: methods to be invoked in an asynchronous manner can not have primitive return types.

A common workaround for this problem consists in introducing own wrapper classes for primitive types, which define methods corresponding to the operators that apply to them [19]. Introducing own wrapper classes can also help preventing the modification of Java system classes such as the standard wrapper classes, which might, should they be passed on and exploited, infringe license terms.

Note that the automatic boxing/unboxing of values of primitive types now in Java 1.5 only slightly alleviates the restricted *completeness*. Values of primitive types
may for instance be boxed automatically by objects, but since this occurs transparently to the programmer there is no way of proxifying these objects in many situations.

Another inherited limitation is the impossibility of proxifying arrays. More precisely, one cannot proxify an array only by creating a proxy for every object it contains rather than for the entire array itself. In the case of an array of dimension larger or equal to 2, this approach must be applied recursively. The only remaining limitation is the impossibility of giving an array a virtual size by overriding its length attribute.

9.4. Modularity

As mentioned there are many ways in which code can be transformed for proxification. The $T_{ax}$ scheme introduced in Section 4.4 achieves modularity in that transformations performed on a class $C$ only depend on $C$ itself. This has obvious advantages when performing these transformations upon class loading. Below we present an alternative earlier design $T_{ax}'$ [17], starting from the example used in Section 4.4.

Original classes: $\xrightarrow{T_{ax}'}$ Transformed classes:

Original code (left) and the code resulting from replacing those lines with corresponding transformations (right) in the original code have the same effect, and in particular the same as $T_{ax}$.
The main difference becomes apparent through class C2. No access methods are created here for field \( s \) unlike in \( T_\alpha \). Corresponding access with a static type \( C2 \) happens on the methods generated for \( C1 \). In other terms, a field access method name is not used to convey static types, but the type declaring the field. In other terms, the lookup of the location of the field occurs before execution. While this solution \( T_\alpha' \) has slight benefits over \( T_\alpha \) in terms of footprint as well as execution, it is not modular. The loading procedure for a class \( C \) might have to be interrupted when transforming field accesses in order to perform lookups in other classes. In contrast, the static type for a field access — as used by \( T_\alpha \) — is readily available in the byte-code. As a consequence, \( T_\alpha' \) more easily breaks when class loaders perform alternative transformations on the same code.

To illustrate the general nature of our formalism introduced in Section 5, Figure 27 presents the necessary modified rules for expressing \( T_\alpha' \) in PJ\( \alpha \). More precisely, the only changes to PJ\( \alpha \) consist in a handful of rules \([X']\) replacing the corresponding rules \([X]\). The transformation \( T_\alpha' \) is identical to \( T_\alpha \) modulo the two rules for transforming location-unspecific field read and write terms. \( T_\alpha' \) is identical to \( T_\alpha \) up to three high-level transformation rules for classes. \( G_\alpha[[T f (C)]] \) is straightforwardly reused.

Note that the reduction rules for field reads and writes remain valid (though they could be simplified as the location of the field is already known), just like the type checking rules for location-unspecific field access terms. Theorems 5 and 6 for subject reduction and progress respectively can be proven for this version of proxies for classes.

10. Related Work

The work closest to ours can be found in the areas of reflection and transparent distribution. We discuss in the following corresponding related efforts. While some of these approaches yield similar functionalities as our uniform proxies when it comes to the proxification of classes and fields, others lack uniform proxification for those features. None of the existing work described in literature enjoys a formal treatment, and none integrates with Java's standard support for dynamic proxies.
10.1. Java Reflection

Various proposals extend the reflective capabilities of Java in a generic manner. More exhaustive surveys of reflective extensions to Java can be found in [72] and [13].

10.1.1. Kava

Kava [72] is a general extension for behavioral reflection, relying on a specific user class loader to modify classes at load-time. Kava follows an *envelopment* approach (unlike its proxy-based predecessor Dalang [71]), in the sense that hooks are added *around* method invocations and field accesses, to pass control to the meta-level.

In the context of dynamic proxies for classes, such an approach could be adapted to transform code (source code for readability) as follows (by omitting exceptions etc.):

```
\Gamma \vdash t : C \\
\Gamma \vdash f(C) = \mathbf{[\ldots, T_0 f(C_0) \ldots]} \\
\Gamma \vdash t : C \\
\Gamma \vdash f(C) = \mathbf{[\ldots, T_0 f(C_0) \ldots]}
```

```
\Gamma \vdash t : C \\
\Gamma \vdash t : T \quad T \leq T_0 \\
\Gamma \vdash f(C) = \mathbf{[\ldots, T_0 f(C_0) \ldots]}
```

Figure 26: Alternative rules for expressing $T_{\text{inv}}$ in $PJ_{\text{inv}}$.

10.1. Java Reflection

Various proposals extend the reflective capabilities of Java in a generic manner. More exhaustive surveys of reflective extensions to Java can be found in [72] and [13].

10.1.1. Kava

Kava [72] is a general extension for behavioral reflection, relying on a specific user class loader to modify classes at load-time. Kava follows an *envelopment* approach (unlike its proxy-based predecessor Dalang [71]), in the sense that hooks are added *around* method invocations and field accesses, to pass control to the meta-level.

In the context of dynamic proxies for classes, such an approach could be adapted to transform code (source code for readability) as follows (by omitting exceptions etc.):
Original code:  \[ \xrightarrow{T} \] Transformed code:

```java
C c = ...;
String f = c.bar;
if (c instanceof ProxyType)
    try {
        Field f = C.class.getField("bar");
        f = (String) Proxy.getAccessHandler(c).get(c, f);
    } catch (Exception ex) {} else f = c.bar;
```

This seminal approach enables the uniform interception of any method invocations and field accesses, including class methods and fields. Just like in our approach, the expensive calls to the core reflection API can be reduced by storing such meta-objects, once retrieved, for further use. In the above approach, this however either leads to the scattered addition of corresponding fields to all call-site classes or requires the use of static fields in a globally visible class. In our approach, such fields are nicely encapsulated in proxy classes, where they are shared yet confined.

The use of an instrumented user class loader can be bypassed (see Section 6.1), thus putting the uniform application of reflection at stake [71]. Just like in our case, transformations affect classes reflected upon (e.g., invocations to own instances) as well as classes using former classes (e.g., invoking instances of classes reflected upon). The advantage of a user class loader is inversely portability with respect to the virtual machine, and the approach might offer benefits for providing identity transparency. Most importantly in the present context though, the approach above does not integrate with the current standard support in Java.

10.1.2. Javassist

Javassist [14] is another extension to Java reflection, promoting load-time structural reflection. Javassist offers a core API operating at byte-code level, and a more high-level API providing useful "macros" built on the former one (e.g., addition or modification of methods), including a specific class loader for performing load-time class instrumentation.

Javassist is extremely general and powerful, and has many potential applications. Behavioral reflection is in fact only one of these instantiations, obtained by wrapping methods. That is, shifts to the meta-level are achieved by inserting hooks into the bodies of methods to be reflected upon rather than around the invocations to them as in Kava. This scheme, in contrast to Kava, establishes a clear equivalence between the classes reflected upon and the classes that have to be modified, i.e., loaded with Javassist's specific class loader. Since this scheme can not be extended to field accesses, Javassist proceeds similarly to our approach in that case by replacing field accesses by invocations, however to class (static) methods.

The general applicability of Javassist has been illustrated by realizing binary code adaptation [37], aspect-oriented programming [35], or a form of synchronous RMI without static proxy generation which has certainly inspired the present support for dynamic proxies in Java. Javassist has been used to implement a first prototype of the
transformations presented in this paper.

10.2. Transparent Distribution

Among specific related work are several approaches for transparent distribution.

10.2.1. Addistant

Addistant [65] is another application of Javassist. Based on the experiences made when implementing RMI in Javassist, Addistant aims at the distribution of “legacy” Java programs, that is, Java programs developed without distribution in mind.

Four different ways of modifying a class to reflect the possibly remote location of certain of its instances are discussed. In the case of a class $C$ whose instances are all remote, the class can for instance be replaced by a proxy class which can assume the name $C$. An extension approach termed subclass approach is also discussed. The problems with final classes and methods, as well as constructors, are pointed out, unlike the cases of private methods and field accesses.

10.2.2. J-Orchestra

J-Orchestra [67, 68] aims at automatic distribution of Java applications just like Addistant. Several of the issues observed in the context of Addistant, such as the difficulties of extending final classes, intercepting field accesses or invocations to private methods, or handling of creation and identity are discussed and solutions offered. Just like Addistant, J-Orchestra allows for mixing of approaches for interception, notably the replacement approach outlined above and the renaming approach [65, 67] where the client class(es) are modified to use a proxy class instead of the original class $C$ if $C$ can not be modified. However, J-Orchestra provides a more differentiated treatment in the case of the latter approach, as it uses combinations of unmodifiable client classes and respective unmodifiable referred classes to guide program partition rather than disallowing the migration of any instance of an unmodifiable class in the presence of a single unmodifiable client class.

The solution proposed for fields consists in replacing something like $\text{o1.a_field}++$ by $\text{o1.set_a_field(o1.get_a_field()+1)}$ [67], which does not support field hiding as is standard in Java. The alteration of visibility modifiers is also performed, though with different outcome and semantics as in our case: even protected methods are made public if they have to be invoked remotely, thus lending a specific meaning to visibility modifiers in scenarios with physically distributed components (as opposed to their usual interpretation with respect to the logical distribution of components, e.g. different classes, different packages). Along the same lines, static fields are viewed as globally unique across nodes, and respective classes thus undergo specific transformations to achieve synchronization. Similarly, class and proxy class instantiation are analyzed thoroughly from a distribution standpoint.

---

9 The corresponding unsafe example has been removed from the extended report together with detailed treatment of the transformation [68].
10.2.3. ProActive

ProActive, a descendant of Java// (“Java parallel”), aims at providing features for transparent distributed or parallel execution of Java programs [12] just like Addistant and J-Orchesttra. ProActive advocates the use of implicit futures though to decouple remotely interacting components, through proxies obtained at run-time by manually instantiating generic proxy classes part of the ProActive libraries. ProActive could thus definitely directly benefit from uniform proxies. A formal semantics for ProActive [11] focuses on concurrency semantics of future method invocations.

10.2.4. RuggedJ

RuggedJ [45, 46] is a more recent effort towards transparent distribution of Java programs. Among the novelties in RuggedJ is a closer investigation of the options for dealing with system classes [46], and a thorough weighing of pros and cons for the presented options. To avoid exposing the changes performed on classes through reflection, RuggedJ includes its own introspection classes.

Distribution in RuggedJ is dealt with by byte-code transformations, consisting in generating an interface $I_C$ for each original class $C$ such that different classes $C_1,...,C_q$ can implement such an interface $I_C$ but provide different functionalities for instances of $C$ depending on the location (e.g., proxy to local object, proxy to remote object, actual implementation). The generated interfaces exhibit subtyping relationships following the inheritance relationships among their respective classes. The transformations described for dealing with private methods [45] (as well as for methods which are protected or have package visibility) consist in making these public to allow for them to appear in the generated interfaces. This can introduce accidental overriding if a class $C$ and its super-class $C'$ declare private methods with equivalent signatures. Retaining the invokespecial byte-code in callees to avoid a dynamic dispatch for originally private methods is impossible, as the introduction of an interface requires the invokeinterface byte-code to be used which invariably leads to a dynamic dispatch. This has been remediated in RuggedJ based on our insights presented herein [44].

10.3. Byte-Code Manipulation Frameworks and Mock Objects

Several frameworks have been proposed to manipulate Java byte-code more conveniently. The Code Generation Library (cglib) [57] is a powerful library used to manipulate Java classes. In particular, it has been used to implement interfaces at run-time, thus providing the same functionality as the original dynamic proxies in Java. cglib builds on the popular Byte Code Engineering Library (BCEL) [3]. Both cglib and BCEL can be typically used to implement the code transformations discussed in this paper. ASM [49] is another byte-code manipulation library, which supports generation, transformation and analysis of compiled Java classes – however without support for tapping into the class loading process. Together with a user class loader which modifies classes to achieve a uniform virtual object model, above-mentioned frameworks can be used to implement transformations as described herein. As mentioned user class loaders can be bypassed and do not apply to system classes, thus our implementation has favored sacrificing independence of our mechanisms from the runtime environment by integrating them more deeply to achieve safety.
PowerMock [] is a framework for implementing mock objects used for testing. In contrast to other frameworks for Java, PowerMock similarly relies on a user class loader and byte code transformations to extend the scope of proxification.

11. Conclusions

This paper has presented an approach to broadening the scope of Java’s own concept of dynamic proxies.

The solution presented in this paper neither makes use of a specific compiler nor relies on dispatch instrumentation in the virtual machine, but uses a set of transformations performed at class loading. For instance, to be able to intercept field accesses we have presented and formalized a scheme for transforming such accesses to invocations of automatically generated getter/setter methods – a general transformation scheme whose applicability is not limited to the generation of dynamic proxies and Java.

We have discussed tradeoffs of proxification, such as between completeness, safety, and security. Given the generic applicability of the proxy abstraction, the set of potential applications which can benefit from our extension is unlimited. We have illustrated our uniform proxies by implementing future invocations both transparently and safely, further dividing transparency into different aspects and determining which of these aspects are actually desirable in what contexts. We have investigated further application scenarios of proxies, pointing out that when transparency is a main concern, structural conformance is best implemented with more specific and inherent language support, and proxies might not be worthwhile for implementing scenarios following wrapper and decorator design patterns, especially when considering the incurred overhead of method invocation reification and reflective execution. Interestingly, these scenarios are among the most frequently mentioned cases for dynamic proxies, and non-experts sometimes confound the latter two altogether with proxies because of their close relationships to the proxy pattern [23].

The main remaining shortcomings of our extended proxy mechanism are a consequence of Java’s hybrid type system. In fact, it’s not surprising that the absence of a uniform object model makes it hard to achieve a uniform proxy model. Hence, it would be interesting to see how one could combine our approach with a uniform object model such as the one promoted by Kava [4].

To increase portability of our implementation, we plan on shifting our transformations to the JVM tool interface (i.e., the java.lang.instrument package) introduced in Java 1.5 as soon as its implementation in Jikes completes.

We believe that our work could also be extended to Microsoft’s .NET platform [66], which proposes a closely related concept of dynamic proxies with nearly the same limitations as in Java. For instance, field accesses can not be intercepted either, which is however counterbalanced by the fact that types in .NET languages such as C# can declare properties, a form of fields with inherent support for getter/setter methods.

---

10 Not to be confused with the Kava approach to behavioral reflection [72], cf. Section 10.1.1.
12. Acknowledgements

The authors would like to thank Christian Damm, Sébastien Baehni, any anonymous reviewers who commented on subsets of the material of this article in the past, as well as all those who contributed to the implementation of this work. We also thank Adam Welc and Phil McGachey for insightful discussions on their respective research efforts related to the present work.

References


[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


