Java Proof Linking with Multiple Classloaders

Philip W. L. Fong
pwfong@cs.sfu.ca

Robert D. Cameron
cameron@cs.sfu.ca

SFU CMPT TR 2000-04
ftp://fas.sfu.ca/pub/cs/TR/2000/

August 23, 2000

Abstract

The Proof Linking Architecture was proposed as a framework for conducting modular verification in the presence of lazy, dynamic linking. The model was instantiated to modularize Java bytecode verification in a simplified Java run-time environment, in which there was a single classloader. This paper analyzes the interaction between proof linking and lazy, dynamic linking in the setting of multiple classloaders. It turns out that a systematic, straightforward set of extensions to the original model is sufficient to make proof linking work with multiple classloaders. This demonstrates that the proof linking idea is applicable to realistic mobile code environments.

1 Introduction

In our previous work [1], we proposed the Proof Linking Architecture as a framework for conducting modular verification in the presence of lazy, dynamic linking. Modularization of verification procedures results in mobile code architectures that are easier to comprehend, maintain and verify. It also enables remote verification and protocol interoperability.

The abstract framework was instantiated to modularize Java bytecode verification [3]. In order to focus on the interplay between incremental proof linking and lazy, dynamic linking, we instantiated the framework in a simplified Java run-time environment, in which there was only one classloader. Under such assumption, we successfully modularized Java bytecode verification, and the correctness of the resulting verification scheme was carefully established.

However, in a standard Java Virtual Machine (JVM), multiple classloaders exist for namespace partitioning [2]. A class defined by one classloader is distinct from another defined by a different classloader, even though the two classes may share the same name. It is unobvious whether the proof linking scheme still applies in an environment with multiple classloaders. This paper analyzes the interaction between proof linking and lazy, dynamic linking in the setting of multiple classloaders. It turns out that a systematic, straightforward set of extensions to the original model is sufficient to

make proof linking work with multiple classloaders. This demonstrates that the proof linking technique is applicable to realistic mobile code environments and is orthogonal to Java's delegation-style classloading.

This paper is a proper sequel to our original work. As such, it assumes that the readers are familiar with the notions defined in the original paper.

2 Multiple Classloaders

As we have briefly mentioned, both the model and the implementation in our original work assumes that there is only a single classloader. In standard Java platforms, multiple namespaces can be created by defining multiple classloaders. A Java class is then identified not only by its name, but by both its name and the classloader in which the class is defined. Formally, when a Java application attempts to load a class C with a given name X by a classloader L_i, *the initiating classloader* of C, L_i may delegate the classloading task to another classloader, which, in turn, might delegate the task to yet another classloader. The classloader L_d that eventually loads and defines C is said to be its *defining classloader*. C is uniquely identified by the pair $\langle X, L_d \rangle$. We also write $X^{L_i} \mapsto \langle X, L_d \rangle$ to indicate the fact that L_i initiates the loading of $\langle X, L_d \rangle$.

When a symbolic reference Y is resolved in a class $\langle X, L \rangle$, the classloader L will be used as the initiating classloader for class Y. Doing so guarantees that the loading of classes referenced in class $\langle X, L \rangle$ is consistently initiated by the same classloader that defines the class.

For more details on Java's classloading scheme, consult Consult the JVM specification [3, Chapter 5] for the official description of Java's classloading scheme and Liang and Bracha's paper [2] for the underlying design rationale.

3 Java Proof Linking for Multiple Classloaders

3.1 Overview of the Solution Approach

A naive attempt to account for the complexity introduced by multiple classloaders would be to replace each class reference Y appearing in commitments and obligations by a classname-classloader pair $\langle Y, K \rangle$, where K is the defining classloader of the referenced class. However, this naive approach would not work. Suppose that the commitment or obligation mentioning Y is generated when a class $\langle X, J \rangle$ is being verified. There is no guarantee that the class designated by the class symbol Y has already been properly loaded before class $\langle X, J \rangle$ is verified (recall that our goal is to avoid recursive classloading). In the adversarial case, the defining classloader for Y is simply not known yet, and so the naive approach will break down for the obvious reason.

Fortunately, the initiating classloader for class Y is already known. When the class $\langle X, J \rangle$ is verified, and thereby generating the commitment or obligation involving class symbol Y, the defining classloader J for X is already known. Java semantics dictates that this classloader J will be used as the initiating classloader for all the classes referenced in $\langle X, J \rangle$. That is, all the classes appearing in the commitments and obligations introduced by the verification procedure have J as their initiating

load $\langle X, J\rangle$	Define a class with name X and defining classloader J. This operation assigns a physical address, denoted by $\langle X, J\rangle$, to a loaded classfile.
verify $\langle X, J\rangle$	Perform modular verification on the classfile of the loaded class $\langle X, J\rangle$.
bind X^L **to** $\langle X, J\rangle$	Bind the class symbol X in the namespace of classloader L to the loaded class $\langle X, J\rangle$. That is, classloader L becomes an initiating classloader of X.
endorse $\langle X, J\rangle$	Endorse the loaded class $\langle X, J\rangle$ for resolution.
endorse $\langle X{::}M(S), J\rangle$	Endorse the loaded member $\langle X{::}M(S), J\rangle$ for resolution.
resolve Y **in** $\langle X, J\rangle$	Resolve the class symbol Y in loaded class $\langle X, J\rangle$.
resolve $Y{::}M(S)$ **in** $\langle X, J\rangle$	Resolve the member symbol $Y{::}M(S)$ in loaded class $\langle X, J\rangle$.

Figure 1: Linking Primitives for Java

classloader. Consequently, class references occurring in commitments and obligations should be represented by the classnames and their corresponding initiating classloaders. The following extensions to our original scheme are required:

1. The binding of a class $\langle X, L_d\rangle$ to a name X^{L_i} in an initiating classloader L_i should be modeled explicitly as a linking primitive, "**bind** X^{L_i} **to** $\langle X, L_d\rangle$".
2. The above primitive should assert the commitment $X^{L_i} \mapsto \langle X, L_d\rangle$.
3. The rules in the initial theory should be updated to make use of the binding commitments ($\mapsto$) for explicitly resolving the occurrences of X^{L_i} in commitments and obligations to $\langle X, L_d\rangle$.

With appropriate refinement to the linking strategy and the initial theory, the above scheme will support proof linking in the presence of multiple classloaders.

3.2 Linking Primitives

We begin the discussion by looking at the extended set of linking primitives in Figure 1. We have introduced two changes to the original primitive set [1, Sec. 4.1]:

1. In the original work, a code unit is designated by a classname. This no longer works because of the presence of multiple classloaders. The primitives **load**, **verify**, **endorse** and **resolve** are adapted to refer to loaded classes (i.e. an ordered pair of a classname and a defining classloader) instead of classnames.
2. A new family of **bind** primitives is introduced. It models the explicit binding of loaded classes to symbols defined in the local namespace of a classloader. When the JVM binds the loaded class $\langle X, J\rangle$ to the symbol X in an initiating classloader

L, the primitive "**bind** X^L **to** $\langle X, J\rangle$" will be executed. It is assumed that the JVM will execute at most one "**bind** X^L **to** $\langle X, J\rangle$" for each symbol X in classloader L.

In addition, we no longer consider the **use** primitive found in our previous work, as doing so does not offer further insight in Java proof linking.

3.3 Static Type Rules

We then examine the static type rules formulated in our previous work[1] [1, Figure 6]. To reason with loaded classes in multiple namespaces instead of static classnames in a single namespace, we begin with the naive approach mentioned in Section 3.1, and later enrich the scheme with our proposed solution approach. We uniformly replace classnames with loaded class notations. For example, we replace the type rule below:

```
subclass(X, X).
subclass(X, Y) :-
   extends(X, Z), subclass(Z, Y).
```

with a new rule of the following form:

```
subclass(⟨X, J⟩, ⟨X, J⟩).
subclass(⟨X, J⟩, ⟨Y, K⟩) :-
   extends(⟨X, J⟩, ⟨Z, L⟩), subclass(⟨Z, L⟩, ⟨Y, K⟩).
```

The reformulation[2] of rules is entirely mechanical. A list of all the reformulated rules is found in Figure 2. The evaluation of the static type rules above requires ability to evaluate the query forms in Figure 3, a topic to which we will turn next.

3.4 Commitment Assertion

Suppose that a classfile with classname X is being verified, and that X extends a class with name Y. We would then want to assert a commitment specifying this subclassing relationship. However, at verification time, the defining classloader K for the superclass Y is not known yet, so the commitment cannot be phrased in terms of $\langle Y, K\rangle$. Moreover, because the actual verification of the classfile may happen remotely, even the defining classloader J for the subclass X might not be known at verification time. As a result, the commitments collected by the verification procedure cannot be phrased in terms of $\langle X, J\rangle$. To deal with this, the bytecode verification procedure instead formulates the commitment:

```
extends(this, Y)
```

The relative reference `this` represents the class being verified. A list of all the commitments that a bytecode verification procedure should generate can be found in Figure 4. They are straightforward reformulation of the commitments found in our original work [1, Figure 4].

[1] In this paper, we have reformatted the datalog-styled notations for commitments and obligations into a style with structures and operators. We believe that doing so improves readability without altering the essence of the scheme.

[2] In fact, the original rule can be used here without change. We explicitly draw attention to the difference in argument types of the predicate symbols here.

```
implementable([]).
implementable([⟨X, J⟩ | T]) :-
   interface(⟨X, J⟩), implements(⟨X, J⟩, Is),
   implementable(Is), implementable(T).

subclassable(⟨'java/lang/Object', bootstrap_classloader⟩).
subclassable(⟨X, J⟩) :-
   class(⟨X, J⟩), non_final(⟨X, J⟩), implements(⟨X, J⟩, Is), implementable(Is),
   extends(⟨X, J⟩, ⟨Y, K⟩), subclassable(⟨Y, K⟩).

subclass(⟨X, J⟩, ⟨X, J⟩).
subclass(⟨X, J⟩, ⟨Y, K⟩) :-
   extends(⟨X, J⟩, ⟨Z, L⟩), subclass(⟨Z, L⟩, ⟨Y, K⟩).

throwable(⟨X, J⟩) :-
   subclass(⟨X, J⟩, ⟨'java/lang/Throwable', bootstrap_classloader⟩).

transitively_implements(⟨X, J⟩, ⟨Y, K⟩) :-
   implements(⟨X, J⟩, Is), member(⟨Y, K⟩, I).
transitively_implements(⟨X, J⟩, ⟨Y, K⟩) :-
   implements(⟨X, J⟩, Is), member(⟨Z, L⟩, Is),
   transitively_implements(⟨Z, L⟩, ⟨Y, K⟩).

superinterface(⟨X, J⟩, ⟨X, J⟩) :-
   interface(⟨X, J⟩).
superinterface(⟨Y, K⟩, ⟨X, J⟩) :-
   subclass(⟨X, J⟩, ⟨Z, L⟩), transitively_implements(⟨Z, L⟩, ⟨Y, K⟩).

assignment_compatible(⟨X, J⟩, ⟨Y, K⟩) :- subclass(⟨X, J⟩, ⟨Y, K⟩).
assignment_compatible(⟨X, J⟩, ⟨Y, K⟩) :- superinterface(⟨Y, K⟩, ⟨X, J⟩).

accessible_member(⟨Y::M(S), K⟩, _, _) :- public_member(⟨Y::M(S), K⟩).
accessible_member(⟨Y::M(S), K⟩, ⟨X, J⟩, _) :- protected_member(⟨Y::M(S), K⟩),
   package(⟨P, L⟩, ⟨X, J⟩), package(⟨P, L⟩, ⟨Y, K⟩).
accessible_member(⟨Y::M(S), K⟩, ⟨X, J⟩, ⟨Z, L⟩) :- protected_member(⟨Y::M(S), K⟩),
   subclass(⟨X, J⟩, ⟨Y, K⟩), subclass(⟨Z, L⟩, ⟨X, J⟩).
accessible_member(⟨Y::M(S), K⟩, _, _) :- default_member(⟨Y::M(S), K⟩).
accessible_member(⟨Y::M(S), K⟩, _, _) :- private_member(⟨Y::M(S), K⟩).
```

Figure 2: Static Type Rules

`class(`$\langle X, J\rangle$`)`	`member(`$\langle X::M(S), J\rangle$`)`
`interface(`$\langle X, J\rangle$`)`	`public_member(`$\langle X::M(S), J\rangle$`)`
`non_final(`$\langle X, J\rangle$`)`	`protected_member(`$\langle X::M(S), J\rangle$`)`
`package(`$\langle P, L\rangle$`,` $\langle X, J\rangle$`)`	`default_member(`$\langle X::M(S), J\rangle$`)`
`extends(`$\langle X, J\rangle$`,` $\langle Y, K\rangle$`)`	`private_member(`$\langle X::M(S), J\rangle$`)`
`implements(`$\langle X, J\rangle$`,` Is`)`	

Figure 3: Foundational Queries

`class(this)`: The classfile defines a class.

`interface(this)`: The classfile defines an interface.

`non_final(this)`: The classfile is not declared to be final.

`package(`P`,this)`: The class belongs to package P.

`extends(this,`Y`)`: The direct superclass is Y.

`implements(this,` L`)`: The list of direct superinterfaces are L.

`member(this::`$M(S)$`)`: $M(S)$ is a member of the class.

`public_member(this::`$M(S)$`)`: $M(S)$ is a public member of the class.

`protected_member(this::`$M(S)$`)`: $M(S)$ is a protected member of the class.

`default_member(this::`$M(S)$`)`: $M(S)$ is a default member of the class.

`private_member(this::`$M(S)$`)`: $M(S)$ is a private member of the class.

`relevant(`Y`,this::`$M(S)$`)`: The class symbol Y is relevant to the endorsement of the method $M(S)$. See Section 3.6 for detail.

Figure 4: Commitments formulated by a bytecode verification procedure.

`bind_class_list([],[])@`$\langle X, J\rangle$.
`bind_class_list([`Y` | `H`],[`$\langle Y, K\rangle$` | `I`])@`$\langle X, J\rangle$:-
    $Y^J \mapsto \langle Y, K\rangle$,
    `bind_class_list(`H`,`I`)@`$\langle X, J\rangle$.

Figure 5: Rules for Resolving a List of Symbols

```
class(⟨X, J⟩) :-
   class(this) @ ⟨X, J⟩.

interface(⟨X, J⟩) :-
   interface(this) @ ⟨X, J⟩.

non_final(⟨X, J⟩) :-
   non_final(this) @ ⟨X, J⟩.

package(⟨P, J⟩, ⟨X, J⟩) :-
   package(P, this) @ ⟨X, J⟩.

extends(⟨X, J⟩, ⟨Y, K⟩) :-
   extends(this, Y) @ ⟨X, J⟩,
   Y^J ↦ ⟨Y, K⟩.

implements(⟨X, J⟩, I) :-
   extends(this, H) @ ⟨X, J⟩,
   bind_class_list(H, I) @ ⟨X, J⟩.

member(⟨X::M(S), J⟩) :-
   member(this::M(S)) @ ⟨X, J⟩.

public_member(⟨X::M(S), J⟩) :-
   public_member(this::M(S)) @ ⟨X, J⟩.

protected_member(⟨X::M(S), J⟩) :-
   protected_member(this::M(S)) @ ⟨X, J⟩.

default_member(⟨X::M(S), J⟩) :-
   default_member(this::M(S)) @ ⟨X, J⟩.

private_member(⟨X::M(S), J⟩) :-
   private_member(this::M(S)) @ ⟨X, J⟩.
```

Figure 6: Translation Rules for Resolving Symbols in Commitments

When the commitments are actually asserted into the commitment database by the "**verify** $\langle X, J\rangle$" primitive, the defining classloader J for class X is known. Therefore, whenever "**verify** $\langle X, J\rangle$" asserts a commitment p, it tags the commitment as p `@` $\langle X, J\rangle$. For example, the commitment above will be asserted as:

```
extends(this, Y) @ ⟨X, J⟩
```

Similar tagging is systematically applied to all the commitments in Figure 4 when they are actually asserted into the commitment database.

The **bind** primitives are another source for commitments. Whenever a "**bind** X^L **to** $\langle X, J\rangle$" primitive terminates, it asserts the commitment "$X^L \mapsto \langle X, J\rangle$". These facts will be used for explicit resolution of symbols in obligations and queries. In order to facilitate resolving lists of class symbols, we introduce the rule in Figure 5 into the initial theory.

To evaluate the queries in Figure 3 by the commitments asserted by the **verify** and **bind** primitives, we need additional translation rules in our initial theory. For example, to check if "`extends(`$\langle X, J\rangle$`,` $\langle Y, K\rangle$`)`" is true, we can make use of the following translation:

```
extends(⟨X, J⟩, ⟨Y, K⟩) :-
   extends(this, Y) @ ⟨X, J⟩,
   Y^J ↦ ⟨Y, K⟩.
```

The rule basically looks up the corresponding tagged commitment, and then validates binding information by consulting the binding commitments. A similar translation rule for each query form in Figure 3 is need, the formulation of which is mechanical. A list of all such translation rules can be found in Figure 6.

3.5 Obligation Attachment

We reproduce (with minor reformatting) from our previous paper the list of obligations that could be generated by a bytecode verification procedure (Figure 7). In the same manner as the commitments, the **verify** primitive has no way of figuring out the defining classloader for the class symbols appearing in the obligations. We then follow the same strategy and formulate obligations in terms of static classnames, and then tag it with the context in which the obligations are to be evaluated. For example, the **verify** primitive may formulate an obligation of the following form:

$\texttt{subclass(}Y\texttt{,}Z\texttt{)}$

and then tag it with an evaluation context before attachment:

$\texttt{subclass(}Y\texttt{,}Z\texttt{)}\ \texttt{@}\ \langle X, J\rangle$

Similar kind of tagging is systematically applied to all the obligations in Figure 7.

Again, in order to evaluate the obligations above, one has to provide translation rules that transform tagged queries into queries in terms of loaded classes. For example, the following rule is required in the initial theory in order to handle all `subclass/2` queries:

$\texttt{subclass(}Y\texttt{,}Z\texttt{)}\ \texttt{@}\ \langle X, J\rangle$:-
$Y^J \mapsto \langle Y, K\rangle$,
$Z^J \mapsto \langle Z, L\rangle$,
$\texttt{subclass(}\langle Y, K\rangle\texttt{,}\langle Z, L\rangle\texttt{)}$.

The translation rule basically resolves all the symbols in the tagged context, and evaluate a corresponding query in terms of loaded classes. A list of all such translation rules can be found in Figure 8.

If verification is performed remotely, then the defining classloader J in Figure 7 will not be available. As a result, one cannot completely identify the target primitives to which the obligations are attached. However, since all the target primitives are operations on $\langle X, J\rangle$, the class being verified, we could apply an old trick to solve the problem: represent $\langle X, J\rangle$ by the relative reference `this`, and only resolve the constant `this` to $\langle X, J\rangle$ when the JVM executes "**verify** $\langle X, J\rangle$" locally.

3.6 Linking Strategy

We adopt the following convention in presenting ordering constraints. For linking primitives p and q, the constraint "$p < q$" requires that any execution of primitive q should be preceded by an execution of primitive p. The constraint "$p < q$ if g", in which g is a database query, requires that, at the time when primitive q is executed, if goal g is satisfiable, then primitive p must have already been executed.

We impose the following ordering constraints to the linking primitives. Except for the newly introduced *Proper Resolution Property*, the rest are basically refinement to those ordering constraints found in the original paper [1, Sec. 4.1]:

1. **Natural Progression Property:** The natural life cycle of a class $\langle X, J\rangle$ is reflected in the ordering below:

   load $\langle X, J\rangle$ < **verify** $\langle X, J\rangle$ < **bind** X^J **to** $\langle X, J\rangle$
   < **endorse** $\langle X, J\rangle$ < **resolve** Y **in** $\langle X, J\rangle$ < **resolve** $Y::M(S)$ **in** $\langle X, J\rangle$

`?subclassable(`Y`)`
Target: endorse $\langle X, J \rangle$
Intention: Direct superclass Y of X can be subclassed.

`?implementable(`L`)`
Target: endorse $\langle X, J \rangle$
Intention: The list L of direct superinterfaces for X are properly defined interfaces.

`?class(`Y`)`
Target: resolve Y **in** $\langle X, J \rangle$
Intention: Y should be a non-interface class.

`?interface(`Y`)`
Target: resolve Y **in** $\langle X, J \rangle$
Intention: Y should be an interface.

`?member(`$Y::M(S)$`)`
Target: resolve $Y::M(S)$ **in** $\langle X, J \rangle$
Intention: $M(S)$ is a member of Y.

`?throwable(`Y`)`
Target: endorse $\langle X::M(S), J \rangle$
Intention: Identified as relevant to $X::M(S)$, class Y is throwable.

`?subclass(`Y`, `Z`)`
Target: endorse $\langle X::M(S), J \rangle$
Intention: Identified as relevant to $X::M(S)$, Y is a subclass of Z.

`?subclass(`X`, `Y`)`
Target: resolve $Y::M(S)$ **in** $\langle X, J \rangle$
Intention: X is a subclass of Y.

`?superinterface(`Z`, `Y`)`
Target: endorse $\langle X::M(S), J \rangle$
Intention: Identified as relevant to $X::M(S)$, Y has Z as a superinterface.

`?assignment_compatible(`Y`, `Z`)`
Target: endorse $\langle X::M(S), J \rangle$
Intention: Identified as relevant to $X::M(S)$, Y is assignment compatible to Z.

`?accessible_member(`$Y::M(S)$`, `X`, `Z`)`
Target: resolve $Y::M(S)$ **in** $\langle X, J \rangle$
Intention: Asserted when a method $N(T)$ of X is verified. It requires that, Z being relevant to $X::N(T)$, a reference of type Z can be used to access the member $M(S)$ of Y.

Figure 7: Obligations that may be asserted by "**verify** $\langle X, J \rangle$"

```
subclassable(Y) @ ⟨X, J⟩ :-
    Y^J ↦ ⟨Y, K⟩,
    subclassable(⟨Y, K⟩).

implementable(I) @ ⟨X, J⟩ :-
    bind_class_list(I, H) @ ⟨X, J⟩
    implementable(H).

class(Y) @ ⟨X, J⟩ :-
    Y^J ↦ ⟨Y, K⟩,
    class(⟨Y, K⟩).

interface(Y) @ ⟨X, J⟩ :-
    Y^J ↦ ⟨Y, K⟩,
    interface(⟨Y, K⟩).

member(Y::M(S)) @ ⟨X, J⟩ :-
    Y^J ↦ ⟨Y, K⟩,
    interface(⟨Y::M(S), K⟩).

throwable(Y) @ ⟨X, J⟩ :-
    Y^J ↦ ⟨Y, K⟩,
    throwable(⟨Y, K⟩).

subclass(Y, Z) @ ⟨X, J⟩ :-
    Y^J ↦ ⟨Y, K⟩,
    Z^J ↦ ⟨Z, L⟩,
    subclass(⟨Y, K⟩, ⟨Z, L⟩).

superinterface(Z, Y) @ ⟨X, J⟩ :-
    Y^J ↦ ⟨Y, K⟩,
    Z^J ↦ ⟨Z, L⟩,
    superinterface(⟨Z, L⟩, ⟨Y, K⟩).

assignment_compatible(Y, Z) @ ⟨X, J⟩ :-
    Y^J ↦ ⟨Y, K⟩,
    Z^J ↦ ⟨Z, L⟩,
    assignment_compatible(⟨Y, K⟩, ⟨Z, L⟩).

accessible_member(Y::M(S), X, Z) @ ⟨X, J⟩ :-
    X^J ↦ ⟨X, J⟩,
    Y^J ↦ ⟨Y, K⟩,
    Z^J ↦ ⟨Z, L⟩,
    accessible_member(⟨Y::M(S), K⟩, X^J, Z^L).
```

Figure 8: Translation Rules for Resolving Symbols in Obligations

2. **Proper Resolution Property:** The defining classloader of a loaded class is used for resolving the symbolic references of the class. We capture the fact by the following constraint:

$$\textbf{bind } Y^J \textbf{ to } \langle Y, K\rangle < \textbf{resolve } Y \textbf{ in } \langle X, J\rangle$$

Delegation of classloading bottoms out when a classloader defines the requested class. This fact is reflected in the following ordering:

$$\textbf{bind } Y^K \textbf{ to } \langle Y, K\rangle < \textbf{bind } Y^J \textbf{ to } \langle Y, K\rangle$$

3. **Import-Checked Property:** Resolving a symbolic reference requires that the target object is well-defined:

$$\textbf{endorse } \langle Y, K\rangle < \textbf{resolve } Y \textbf{ in } \langle X, J\rangle \qquad \text{if } Y^J \mapsto \langle Y, K\rangle$$
$$\textbf{endorse } \langle Y{::}M(S), K\rangle < \textbf{resolve } Y{::}M(S) \textbf{ in } \langle X, J\rangle \qquad \text{if } Y^J \mapsto \langle Y, K\rangle$$

4. **Subtype Dependency Property:** To establish an obligation concerning a class, type information about its superclasses and superinterfaces might be needed. For example, to establish that the direct superclass Y^J of a loaded class $\langle X, J\rangle$ is subclassable (i.e. `subclassable(`Y`)@`$\langle X, J\rangle$), We require that all superclasses and superinterfaces of $\langle X, J\rangle$ to be loaded, verified and bound before $\langle X, J\rangle$ is used. To address this need, we require that

$$\textbf{bind } Y^L \textbf{ to } \langle Y, K\rangle < \textbf{endorse } \langle X, J\rangle \qquad \text{if } \texttt{subtypedependent(}Y^L\texttt{)@}\langle X, J\rangle$$

where the conditional query is handled by the following rules in the initial theory:

`subtypedependent(`X^J`)@`$\langle X, J\rangle$`.`

`subtypedependent(`Y^L`)@`$\langle X, J\rangle$ `:-`
`  subtypedependent(`Z^K`)@`$\langle X, J\rangle$`,`
$Z^K \mapsto \langle Z, L\rangle$`,`
`  extends(this,`Y`)@`$\langle Z, L\rangle$`.`

`subtypedependent(`Y^L`)@`$\langle X, J\rangle$ `:-`
`  subtypedependent(`Z^K`)@`$\langle X, J\rangle$`,`
$Z^K \mapsto \langle Z, L\rangle$`,`
`  implements(this,`I`)@`$\langle Z, L\rangle$`,`
`  member(`Y`,`I`).`

5. **Referential Dependency Property:** Sometimes, verification of a class Y is needed before we can safely endorse a method $\langle X{::}M(S), J\rangle$. For example, if method $\langle X{::}M(S), J\rangle$ assigns a reference of type Y to a variable of type Z, then Java type rules require Z to be either a superclass or a superinterface of Y. Unless Y is a superclass of X, it is entirely possible that the superclasses and superinterfaces of Y are not verified yet. Consequently, the required supports for the obligation are not necessarily present at the time the obligation is checked, a violation of the Completion Property. In such a case, we say that Y is *relevant* to the endorsing of $\langle X{::}M(S), J\rangle$. We assume that, verification of the bytecode for method $\langle X{::}M(S), J\rangle$ results in the assertion of commitments `relevant(`Y`,this::`$M(S)$`)@`$\langle X, J\rangle$ for all relevant class symbols Y, and we require that:

$$\textbf{endorse } \langle Y, K\rangle < \textbf{endorse } \langle X{::}M(S), J\rangle \qquad \text{if } \texttt{relevant(}Y\texttt{,this::}M(S)\texttt{)@}\langle X, J\rangle$$

That is, we want to verify all relevant classes (plus their superclasses and superinterfaces) before we check the obligations attached to "**endorse** $\langle X{::}M(S), J\rangle$".

3.7 Putting It All Together: An Example

To illustrate how the scheme above works, consider the following example. Suppose class $\langle A, L_1\rangle$ defines a method $M(S)$. Suppose further that $\langle A, L_1\rangle$ has a direct subclass $\langle B, L_2\rangle$, which in turn has a direct subclass $\langle C, L_3\rangle$. Assume that $\langle C, L_3\rangle$ overrides the method $M(S)$.

Say $\langle C{::}M(S), L_3\rangle$ contains an `invokespecial` instruction that delegates the call to $\langle A{::}M(S), L_1\rangle$. The obligation $\texttt{subclass(}C\texttt{, }A\texttt{)}\ @\ \langle C, L_3\rangle$ will be attached to the primitive "**resolve** $A{::}M(S)$ **in** $\langle C, L_3\rangle$" (see Figure 7). When the obligation is checked, the following subgoals will be generated (see Figure 2, 6 and 8):

1. $\texttt{subclass(}C\texttt{, }A\texttt{)}\ @\ \langle C, L_3\rangle$ /* **resolve** $A{::}M(S)$ **in** $\langle C, L_3\rangle$ */
   - 1.1. $C^{L_3} \mapsto \langle C, L_3\rangle$ /* **bind** C^{L_3} **to** $\langle C, L_3\rangle$ */
   - 1.2. $A^{L_3} \mapsto \langle A, L_1\rangle$ /* **bind** A^{L_3} **to** $\langle A, L_1\rangle$ */
   - 1.3. $\texttt{subclass(}\langle C, L_3\rangle\texttt{, }\langle A, L_1\rangle\texttt{)}$
     - 1.3.1. $\texttt{extends(}\langle C, L_3\rangle\texttt{, }\langle B, L_2\rangle\texttt{)}$
       - 1.3.1.1. $\texttt{extends(this, }B\texttt{)}\ @\ \langle C, L_3\rangle$ /* **verify** $\langle B, L_2\rangle$ */
       - 1.3.1.2. $B^{L_3} \mapsto \langle B, L_2\rangle$ /* **bind** B^{L_3} **to** $\langle B, L_2\rangle$ */
     - 1.3.2. $\texttt{subclass(}\langle B, L_2\rangle\texttt{, }\langle A, L_1\rangle\texttt{)}$
       - 1.3.2.1. $\texttt{extends(}\langle B, L_2\rangle\texttt{, }\langle A, L_1\rangle\texttt{)}$
         - 1.3.2.1.1. $\texttt{extends(this, }A\texttt{)}\ @\ \langle B, L_2\rangle$ /* **verify** $\langle B, L_2\rangle$ */
         - 1.3.2.1.2. $A^{L_2} \mapsto \langle A, L_1\rangle$ /* **bind** A^{L_2} **to** $\langle A, L_1\rangle$ */
       - 1.3.2.2. $\texttt{subclass(}\langle A, L_1\rangle\texttt{, }\langle A, L_1\rangle\texttt{)}$

The original obligation is shown as the top-level goal, annotated with "**resolve** $A{::}M(S)$ **in** $\langle C, L_3\rangle$", the primitive to which the obligation is attached. We have also annotated all the innermost subgoals with the primitives that assert their matching commitments (see Figure 4).

The deduction is successful because the commitments required by the innermost subgoals are already asserted at the time the obligation is checked, that is, at the time the time "**resolve** $A{::}M(S)$ **in** $\langle C, L_3\rangle$" is executed. For example, subgoal 1.1 is satisfiable because, according to the Natural Progression Property, the primitive "**bind** C^{L_3} **to** $\langle C, L_3\rangle$" has already been executed. Also, subgoal 1.2 is satisfiable because

$$\begin{aligned}\textbf{bind } A^{L_3} \textbf{ to } \langle A, L_1\rangle &< \textbf{resolve } A \textbf{ in } \langle C, L_3\rangle && \text{(Proper Resolution)}\\ &< \textbf{resolve } A{::}M(S) \textbf{ in } \langle C, L_3\rangle && \text{(Natural Progression)}\end{aligned}$$

The rest of the subgoals are more interesting. Note that $\texttt{subtypedependent(}B^{L_3}\texttt{)}\ @\ \langle C, L_3\rangle$ is satisfiable before "**resolve** $A{::}M(S)$ **in** $\langle C, L_3\rangle$" is executed. By applying the Subtype Dependency Property and other ordering constraints, we deduce

$$\begin{aligned}\textbf{verify } \langle B, L_2\rangle &< \textbf{bind } B^{L_2} \textbf{ to } \langle B, L_2\rangle && \text{(Natural Progression)}\\ &< \textbf{bind } B^{L_3} \textbf{ to } \langle B, L_2\rangle && \text{(Proper Resolution)}\\ &< \textbf{endorse } \langle C, L_3\rangle && \text{(Subtype Dependency)}\\ &< \textbf{resolve } B{::}M(S) \textbf{ in } \langle C, L_3\rangle && \text{(Natural Progression)}\end{aligned}$$

That is, the commitments $\texttt{extends(this, }B\texttt{)}\ @\ \langle C, L_3\rangle$ and $B^{L_3} \mapsto \langle B, L_2\rangle$, generated by "**verify** $\langle B, L_2\rangle$" and "**bind** B^{L_3} **to** $\langle B, L_2\rangle$" respectively, are already in place when

the obligation is checked. Therefore, subgoals 1.3.1.1. and 1.3.1.2. are necessarily satisfiable. Similar reasoning applies to subgoal 1.3.2.1.1. and 1.3.2.1.2.

What we have achieved above is really a skeleton for the proof of Completion, one of the three correctness criteria for proof linking. Detail correctness justification of Java proof linking with multiple classloaders will be the topic of the next section.

4 Correctness

Recall that, given a well-defined linking strategy, proof linking is correct if we can establish the three correctness conditions: safety, monotonicity and completion [1, Sec. 3.3].

4.1 Consistency of the Linking Strategy

The strict partial order imposed by the linking strategy above is well-defined. To see this, consider the following linearization of the linking primitives:

1. "**load** $\langle X, J\rangle$" for all classnames X and classloaders J
2. "**verify** $\langle X, J\rangle$" for all loaded class $\langle X, J\rangle$
3. "**bind** X^J **to** $\langle X, J\rangle$" for all loaded class $\langle X, J\rangle$
4. "**bind** X^L **to** $\langle X, J\rangle$" for all loaded class $\langle X, J\rangle$ and classloader L such that $J \neq L$
5. "**endorse** $\langle X, J\rangle$" for all loaded class $\langle X, J\rangle$
6. "**endorse** $\langle X::M(S), J\rangle$" for all loaded member $\langle X::M(S), J\rangle$
7. "**resolve** Y **in** $\langle X, J\rangle$" for all class symbols Y and loaded class $\langle X, J\rangle$
8. "**resolve** $Y::M(S)$ **in** $\langle X, J\rangle$" for all member references $Y::M(S)$ and loaded class $\langle X, J\rangle$.

It is easy to check that this linearization satisfies all the ordering constraints imposed by the linking strategy above.

4.2 Safety and Monotonicity

The exact same argument in our previous paper [1, Sec. 4.3] can be applied here to establish safety and monotonicity.

1. **Safety:** Only **verify** primitives generate obligations. A "**verify** $\langle X, J\rangle$" primitive only attaches obligations to "**endorse** $\langle X, J\rangle$", "**endorse** $\langle X::M(S), J\rangle$", "**resolve** Y **in** $\langle X, J\rangle$" and "**resolve** $Y::M(S)$ **in** $\langle X, J\rangle$", all of which are ordered after "**verify** $\langle X, J\rangle$" by the Natural Progression Property. Therefore, once a primitive begins execution, no more unchecked obligation will be attached to it.
2. **Monotonicity:** The rules, commitments and obligations forms a monotonic theory. Once an obligation is satisfied, it will not be contradicted by subsequently asserted commitments.

4.3 Completion

Completion has to be established on an obligation-by-obligation basis. Continuing with our running example in section 3.7, we consider an obligation $\texttt{subclass(}X_0\texttt{, }X_n\texttt{)}\ \texttt{@}\ \langle X_0, L_0\rangle$ that is attached to the primitive "**resolve** $X_n{::}M(S)$ **in** $\langle X_0, L_0\rangle$". Our goal is to show that, if the predicate $\texttt{subclass(}X_0\texttt{, }X_n\texttt{)}\ \texttt{@}\ \langle X_0, L_0\rangle$ eventually becomes provable, then it is provable before the primitive "**resolve** $X_k{::}M(S)$ **in** $\langle X_0, L_0\rangle$" is executed.

Suppose that the obligation $\texttt{subclass(}X_0\texttt{, }X_n\texttt{)}\ \texttt{@}\ \langle X_0, L_0\rangle$ becomes provable at a certain point. Generalizing the proof tree found in Section 3.7, the proof of the obligation contains the following *supports*:

(α-0)	$X_0^{L_0} \mapsto \langle X_0, L_0\rangle$	**bind** $X_0^{L_0}$ **to** $\langle X_0, L_0\rangle$
(γ)	$X_n^{L_0} \mapsto \langle X_n, L_n\rangle$	**bind** $X_n^{L_0}$ **to** $\langle X_n, L_n\rangle$
(β-0)	$\texttt{extends(this, }X_1\texttt{)}\ \texttt{@}\ \langle X_0, L_0\rangle$	**verify** $\langle X_0, L_0\rangle$
(α-1)	$X_1^{L_0} \mapsto \langle X_1, L_1\rangle$	**bind** $X_1^{L_0}$ **to** $\langle X_1, L_1\rangle$
(β-1)	$\texttt{extends(this, }X_2\texttt{)}\ \texttt{@}\ \langle X_1, L_1\rangle$	**verify** $\langle X_1, L_1\rangle$
(α-2)	$X_2^{L_1} \mapsto \langle X_2, L_2\rangle$	**bind** $X_2^{L_1}$ **to** $\langle X_2, L_2\rangle$
(β-2)	$\texttt{extends(this, }X_3\texttt{)}\ \texttt{@}\ \langle X_2, L_2\rangle$	**verify** $\langle X_2, L_2\rangle$
(α-3)	$X_3^{L_2} \mapsto \langle X_3, L_3\rangle$	**bind** $X_3^{L_2}$ **to** $\langle X_3, L_3\rangle$
$\vdots$	$\vdots$	$\vdots$
(β-$(n-1)$)	$\texttt{extends(this, }X_n\texttt{)}\ \texttt{@}\ \langle X_{n-1}, L_{n-1}\rangle$	**verify** $\langle X_{n-1}, L_{n-1}\rangle$
(α-n)	$X_n^{L_{n-1}} \mapsto \langle X_n, L_n\rangle$	**bind** $X_n^{L_{n-1}}$ **to** $\langle X_n, L_n\rangle$

We number the supports as (α-i), (β-i) and (γ). We want to show that the primitives that assert these supports have all been executed prior to the execution of "**resolve** $X_n{::}M(S)$ **in** $\langle X_0, L_0\rangle$". As already explained in Section 3.7, the Proper Resolution Property guarantees that supporting commitment (γ) is already in place. We use induction to show that commitments (α-i) and (β-i) are already asserted when the obligation is checked.

Basis: Commitment (α-0) and (β-0) are already asserted because, by the Natural Progression Property,

$$\textbf{verify}\ \langle X_0, L_0\rangle < \textbf{bind}\ X_0^{L_0}\ \textbf{to}\ \langle X_0, L_0\rangle < \textbf{resolve}\ X_k{::}M(S)\ \textbf{in}\ \langle X_0, L_0\rangle$$

Induction Step: Assume that commitments (α-i) and (β-i) are already in place, for $0 \le i < k$, where $k > 0$. The presence of these commitments enable the query $\texttt{subtypedependent(}X_k^{L_{k-1}}\texttt{)}\ \texttt{@}\ \langle X_0, L_0\rangle$ to be satisfiable. We then have

$$\begin{aligned}
\textbf{verify}\ \langle X_k, L_k\rangle &< \textbf{bind}\ X_k^{L_k}\ \textbf{to}\ \langle X_k, L_k\rangle && \text{(Natural Progression)}\\
&< \textbf{bind}\ X_k^{L_{k-1}}\ \textbf{to}\ \langle X_k, L_k\rangle && \text{(Proper Resolution)}\\
&< \textbf{endorse}\ \langle X_0, L_0\rangle && \text{(Subtype Dependency)}\\
&< \textbf{resolve}\ X_n{::}M(S)\ \textbf{in}\ \langle X_0, L_0\rangle && \text{(Natural Progression)}
\end{aligned}$$

Since the contributors "**bind** $X_k^{L_{k-1}}$ **to** $\langle X_k, L_k\rangle$" and "**verify** $\langle X_k, L_k\rangle$" for respectively (α-k) and (β-k) are already executed, the commitments are present when the obligation is checked.

This concludes the proof of completion for one class of obligations. Completion can be established similarly for the rest of the obligations.

5 Discussion

5.1 Implementation Guidelines

The linking strategy above suggests a natural implementation of proof linking in a JVM with multiple classloaders. In particular, a call to the `defineClass` method of class loader J, with argument X as the expected classname, will execute "**load** $\langle X, J\rangle$" and "**verify** $\langle X, J\rangle$". A call to the `loadClass` method of class loader K, requesting the loading of class X, corresponds to the primitive "**bind** X^K **to** $\langle X, J\rangle$". The primitive "**endorse** $\langle X, J\rangle$" could be executed when the class "$\langle X, J\rangle$" is prepared [3, Sec. 5.4.2]. The primitive "**endorse** $\langle X::M(S), J\rangle$" could be executed right before the method is first resolved. The resolution primitives coincides with regular symbol resolution.

5.2 Comparison with Sun's Linking Strategy

As opposed to Sun's JVM implementation, which postpones bytecode verification until a class is linked, the implementation strategy above performs eager verification, that is, a classfile is verified immediately after it is defined. We have to do so because we want to integrate various "passes" of classfile verification into a single verification primitive. Sun's JVM performs one pass of verification at class definition time, and postpone the second and third passes until link time. This is exactly the arrangement we want to avoid.

A second difference between the implementation above and Sun's existing implementation is that Sun's JVM loads all the superclasses, superinterfaces and relevant classes when a class is defined, while our strategy defers the loading of these classes until the time a class is endorsed (i.e. at link time). However, this difference is not likely to be observable because most class definition is initiated as a direct result of linking activities.

In summary, the net effect of our arrangement is that verification is performed in a more eager manner. However, this is a price one has to pay if we are to have a modular verification procedure that supports interoperability of verification protocols.

5.3 Class Finalization

Our practice of tagging commitments with the contributing loaded class allows us to integrate proof linking into a JVM implementation that supports unloading of classes. According to the JVM Specification, a class is unloaded when its defining classloader becomes unreachable [3, Sec. 2.17.8]. At the point when a loaded class $\langle X, J\rangle$ is unloaded, all commitments tagged with $\langle X, J\rangle$ should be retracted from the commitment database. In addition, all the primitives related to $\langle X, J\rangle$ should be marked as being not executed yet.

6 Conclusion

By introducing only moderate extensions to the original model, we have successfully accounted for the presence of multiple classloaders in Java proof linking. Not only

does this work justify our original decision to abstract away the complexity of multiple classloaders, and to focus on the interplay between incremental proof linking and lazy, dynamic linking, it also demonstrates that the proof linking idea is applicable to realistic mobile code environments.

References

[1] Philip W. L. Fong and Robert D. Cameron. Proof linking: Modular verification of mobile programs in the presence of lazy, dynamic linking. *ACM Transactions on Software Engineering and Methodology*, 2000. To appear. Also available as `http://www.cs.sfu.ca/~pwfong/personal/Pub/tosem2000.ps`.

[2] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, pages 36–44, Vancouver, British Columbia, October 1998. Also available at `http://java.sun.com/people/gbracha/classloader.ps`.

[3] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999. Also available at `http://java.sun.com/docs/books/vmspec`.