Available online at www.sciencedirect.com

ScienceDirect

Theoretical Computer Science 368 (2006) 1–29

Theoretical Computer Science

www.elsevier.com/locate/tcs

A flexible model for dynamic linking in Java and C#

Sophia Drossopoulou[a,*], Giovanni Lagorio[b], Susan Eisenbach[a]

[a] *Department of Computing, Imperial College, London, UK*
[b] *DISI, University of Genova, Italy*

Received 2 April 2005; received in revised form 15 February 2006; accepted 28 February 2006

Communicated by B. Pierce

Abstract

Dynamic linking supports flexible code deployment, allowing partially linked code to link further code on the fly, as needed. Thus, end-users enjoy the advantage of automatically receiving any updates, without any need for any explicit actions on their side, such as re-compilation, or re-linking. On the down side, two executions of a program may link in different versions of code, which in some cases causes subtle errors, and may mystify end-users.

Dynamic linking in Java and C# are similar: the same linking phases are involved, soundness is based on similar ideas, and executions which do not throw linking errors give the same result. They are, however, not identical: the linking phases are combined differently, and take place in different order. Consequently, linking errors may be detected at different times by Java and C# runtime systems.

We develop a non-deterministic model, which describes the behaviour of both Java and C# program executions. The non-determinism allows us to describe the design space, to distill the similarities between the two languages, and to use one proof of soundness for both. We also prove that all execution strategies are equivalent with respect to terminating executions that do not throw link errors: they give the same results.

Keywords: Dynamic linking; Loading; Verifier; Jit-compilation; Types; Operational semantics; Code deployment; Java; C#

1. Introduction

Dynamic linking supports flexible code deployment and update: instead of linking all code before execution, code is linked on the fly, as needed. Thus, the *newest* version of any imported code is always linked, and the most recent updates are automatically available to users *without* the need for *any* action, such as recompilation or import, on their part—provided, of course, that the new version has been installed on the user's machine, or is available in the loader's search space.

* Corresponding author. Tel.: +44 207 494 8368; fax: +44 20 7581 8024.
E-mail addresses: sd@doc.ic.ac.uk (S. Drossopoulou), lagorio@disi.unige.it (G. Lagorio), sue@doc.ic.ac.uk (S. Eisenbach).

0304-3975/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.02.021

Dynamic linking was incorporated into operating systems, e.g., by Multics [44], Unix, and Windows. Dynamic link libraries (DLLs) enable applications running on a single system to share code, thus saving both disk and memory usage. DLLs are linked at runtime, and so, when a DLL is updated, all applications stand to benefit immediately. Java and C# [1] [30] were the first widespread statically typed languages to incorporate dynamic linking into the language design.

A first question connected to dynamic linking is the choice of components to be linked, especially if there are several components with the same name. Choosing a compatible DLL is not always straightforward, and difficulties in managing DLLs led to the term "DLL Hell'' [35]. The .NET architecture claims to have solved this problem with sophisticated systems of versioning and side-by-side components [36]. Java, on the other hand, has a simple approach, whereby it links the first class with a given name found in the classpath; more sophisticated schemes can be implemented through custom class loaders [34].

A second question connected to dynamic linking is the type safety guarantees given *after* choosing components. Breaking type safety jeopardizes the integrity of memory, and ultimately security [22,43]. DLLs do not attempt to guarantee type safety: thus type errors may occur and go undetected, or throw exceptions of an unrelated nature in an unrelated part of the code. Conversely, Java and C# employ verifiers and further mechanisms to guarantee type safety. If the components turn out to be "incompatible'', link related exceptions are thrown, describing the nature of the problem. Thus, although Java and C# do not guarantee the choice of compatible components, they do guarantee type safety and give error messages that signal the source of the problem.

Our study is concerned with how Java and C# tackle the second question, that is, how they guarantee type safety. Dynamic linking in Java and C# are similar: the same linking phases are involved, i.e., loading, verification, offset calculation, and layout determination. Soundness is based on similar ideas: i.e., consistency of the layout and virtual tables, verifying intermediate code, and checking before calculating offsets. Executions which do not throw linking errors always give the same results.

Notwithstanding the similarities, dynamic linking in Java and C# have *some* differences: the linking phases have different granularity, are combined differently and take place in a different order. As a result, linking errors may be detected at different times by Java and C# program executions. [2]

In this paper, we develop a non-deterministic model, which describes the behaviour of both Java and C# programs. We prove preservation properties, i.e., that the dynamic linking phases preserve subtypes, offsets, types of expressions, well-formedness of programs, etc. We believe that such preservation properties were implicitly assumed in the design of dynamic linking. We then prove soundness, i.e., that execution preserves the type of expressions and well-formedness of both program and heap, by means of a subject reduction theorem. We also prove equivalence of execution, i.e., that all executions which do not throw link errors give the same results.

Our model is concerned with the interplay of the phases rather than with the particular phases themselves. It is at a higher level than the Java bytecode or the .NET intermediate language, IL. It abstracts from Java's multiple loaders and .NET assemblies, and describes the verifier as a type checker, disregarding type inference and data flow analysis issues. It models intermediate code as being interpreted, disregarding the difference between JVM bytecode interpretation, and .NET IL code jit-compilation. It represents dynamic linking *not* necessarily as it *is*, but as it is *perceived* by the source language programmer.

On the other hand, in order to describe features salient to dynamic linking, some other aspects of the model are, necessarily, low level. In particular, the model reflects class layout and virtual tables, which, although not part of the programming languages Java and C#, become apparent when considering the effects of fooling verification.

This paper presents further work to that presented at ESOP'03 [18] on flexible models for dynamic linking. Here, we offer a slightly more abstract model, we give some additional explanations, we sketch the proofs in some detail, and we illustrate the formalism through two examples. The rest of this paper is organized as follows: Section 2 introduces Java and C# dynamic linking with an example. Sections 3 and 4 outline and define the model. Section 5 states preservation properties, and soundness of the type system, and sketches the proofs. Section 6 states and sketches the proof of the equivalence of execution strategies. Section 7 concludes. The Appendix contains two further examples illustrating finer points.

---

[1] Actually it is the common runtime system of .NET which deals with many languages rather than just C# that incorporates dynamic linking, but we focus on C# in this paper, as it is most easily compared with Java.

[2] And sometimes they are not exhibited at all.

2. Introduction to the dynamic linking phases

In the presence of dynamic linking, execution can be understood in terms of the following phases:

- evaluation, which is not affected by dynamic linking,
- loading, which reads classes from the environment,
- verification/jit-compilation, which checks type-safety of the code/translates intermediate representations of code to native code,
- laying out, which determines object layout and method tables,
- offset calculation, which replaces references to fields and methods in terms of their signature, through the corresponding offsets.

Phases depend on each other: a class can only be laid out after it has been loaded. The offset of a member from a class may only be calculated after that class has been laid out. When verification/jit-compilation requires some class to extend a further class, it will load the two classes—although [37] suggests a lazier approach of posting constraints instead.

As shown in Table 1, in Java and C# these phases are at different levels of granularity: loading and laying out apply to classes; verification applies to individual method bodies in C#, and to all method bodies of a class in Java; offset calculation applies to individual member access expressions. Also, the phases are organized differently: in Java, offset calculation takes place only just before the particular member is accessed, whereas in C#, offset calculation takes place during jit-compilation. In Java, verification of a class takes place before the first object of that class is created, and involves

Table 1
Dynamic linking phases, granularity and organization

Phase	Granularity		Organization	
	Java	C#	Java	C#
Load	Per class		After superclass loaded	
Verify/jit	Per class	Per method	Before creation of first object	Before invocation, with jit-compilation
Layout	Per class		Before jit-compilation or first member access	
Offset calculation	Per field access/method invocation		Before field access/ method invocation	At jit-compilation

```
class Meal {
    void eat (Penne p){ chew (p); }
    void chew (Pasta p) {
        if (p==null) print(0);
        else print(p.cal);
    }
}

class Food{
    public static void main (String[] args) {
        print (“1’’); Meal m= new Meal ();
        print (“2’’); Penne p= new Penne ();
        print (“3’’); m.eat (null);
        print (“4’’); m.eat (p);
    }
}
```

Fig. 1. Example program.

Table 2
Successful execution of the Meal example—assuming Cls, Fld, Sub

Java phases	Out	C# phases	Out
		calc. offset for main	
verify Food ↪ verify main ↪ check Meal ⩽ Meal ↪ check Penne ⩽ Penne		jit main ↪ check Meal ⩽ Meal ↪ load Meal ↪ lay out Meal ↪ check Penne ⩽ Penne ↪ load Penne; Pasta ↪ calc. offset for void eat (Penne) ↪ lay out Penne ↪ lay out Pasta	
calc. offset for main			
execute main		execute main	
	1		1
lay out Meal ↪ verify Meal ↪ verify void eat (Penne) ↪ check Penne ⩽ Pasta ↪ load Penne; Pasta ↪ verify void chew (Pasta)			
create new Meal object		create new Meal object	
	2		2
lay out Penne ↪ lay out Pasta ↪ verify Pasta ... ↪ verify Penne ...			
create new Penne object		create new Penne object	
	3		3
calc. offset for void eat (Penne)		jit eat (Penne) ↪ check Penne ⩽ Pasta ↪ calc. offset for void chew (Pasta)	
execute eat (Penne)		execute eat (Penne)	
calc. offset for void chew (Pasta)		jit void chew (Pasta) ↪ calc. offset for **int** cal	
execute void chew (Pasta)		execute void chew (Pasta)	
	0		0
	4		4
execute void eat (Penne)		execute void eat (Penne)	
execute void chew (Pasta)		execute void chew (Pasta)	
calc. offset for **int** cal			
	100		100

verification of all methods of that class, whereas in C#, methods are jit-compiled separately, and only before the first execution of that method.

The example from Fig. 1 serves to illustrate these points. The Java classes have been compiled using j2sdk1.4, and executed with the verbose flag set to on. The C# classes have been compiled using the C# compiler version 7.00.9466, and profiling information was obtained using the .NET profiling tool. The complete code in both C# and Java, and instructions on how to produce the behaviour described in this paper, is available at: http://www.doc.ic.ac.uk/∼sue/foodexample.html.

The example consists of classes Meal and Food, compiled in an environment containing previously compiled versions of the classes Pasta and Penne:

class Pasta { **int** cal = 100; },

class Penne **extends** Pasta { }.

Table 3
Execution when ¬ Fld

Java phases	Out	C# phases	Out
		calc. offset for main	
verify Food		jit main ...	
calc. offset for main			
execute main		execute main	
	1		1
lay out Meal ...			
create new Meal object		create new Meal object	
	2		2
lay out Penne ...			
create new Penne object		create new Penne object	
	3		3
calc. offset for void eat (Penne)		jit eat (Penne) ↪ check Penne ⩽ Pasta ↪ calc. offset for void chew (Pasta)	
execute void eat (Penne)		execute void eat (Penne)	
calc. offset for void chew (Pasta)		jit chew (Pasta) ↪ calc. offset for **int** cal ↪ `NoFieldErr`, if ¬ Fld	X
execute void chew (Pasta)			
	0		
	4		
execute void eat (Penne)			
execute void chew (Pasta)			
calc. offset for **int** cal ↪ `NoFieldErr`, if ¬ Fld	X		

Table 4
Execution when ¬ Cls

Java phases	Out	C# phases	Out
		calc. offset for main	
verify Food ↪ verify main ↪ check Meal ⩽ Meal ↪ check Penne ⩽ Penne		jit main ↪ check Meal ⩽ Meal ↪ load Meal ↪ lay out Meal ↪ check Penne ⩽ Penne ↪ load Penne; Pasta ↪ `LoadErr` if ¬ Cls	X
calc. offset for main			
execute main			
	1		
lay out Meal verify Meal ↪ verify void eat (Penne) ↪ check Penne ⩽ Pasta ↪ load Penne; Pasta ↪ `LoadErr` if ¬ Cls	X		

These classes satisfy the following three requirements:

Cls: classes Pasta and Penne are present,
Sub: Penne is a subclass of Pasta,
Fld: Pasta contains a field cal of type **int**,

Table 5
Execution when ¬ Sub

Java phases	Out	C# phases	Out
		calc. offset for main	
verify Food		jit main	
calc. offset for main			
execute main		execute main	
	1		1
lay out Meal ↪ verify Meal ↪ verify void eat (Penne) ↪ check Penne ⩽ Pasta ↪ load Penne; Pasta ↪ VerifErr, if ¬ Sub	X		
		create new Meal object	
			2
		create new Penne object	
			3
		jit void eat (Penne) ↪ check Penne ⩽ Pasta ↪ VerifErr, if ¬ Sub	X

which are crucial for the execution of the method main in Food. Namely, if Cls does not hold then a new Penne object cannot be created. If Sub does not hold, the eat method body cannot be successfully verified, and if Fld does not hold, cal cannot be accessed.

If Cls, Fld and Sub all hold, execution will be successful, and the Java and C# programs will give the same output. This is shown in Table 2. The first and third columns contain the linking phases as they occur in Java or in C#, with their dependencies indicated through the ↪ symbol, e.g., in Java, verification of class Meal requires verification of method eat, which in its turn checks that Pasta ⩽ Penne. The second and fourth columns contain the output from the Java and the C# program executions, e.g., 1, 2, etc.

When Cls, Fld, or Sub does not hold, a link-related exception will be thrown. Although it will be the same exception in both Java and C#, it will be thrown at a different time in execution. Thus, our example demonstrates the following differences:

Offset calculation is "*lazier*'' *in Java*: In our example, ¬ Fld would cause a linking error when attempting to calculate the offset for the field cal from Pasta. In Java this happens before the first attempt to actually access the field, i.e., after printing 4, whereas in C# this happens when jit-compiling the method containing this field access, i.e., after printing 3. This is shown in Table 3, where X indicates an exception.

Subtypes are "*optimistic*" *in Java*: In our example, ¬ Cls could cause a linking error when attempting to load class Pasta or Penne. In Java, because a class is considered a subclass of itself, even if not loaded, verification of main does not require the loading of Penne; and Penne only needs to be loaded when verifying method eat, i.e., after printing 1. In C#, because a class is considered a subclass of itself only if loaded, jit-compilation of main requires loading of Penne, and thus, Penne needs to be loaded even before the beginning of execution. This is shown in Table 4.

Verification is "*lazier*'' *in* C#: In our example, ¬ Sub means that the method eat from class Pasta would not verify. In Java, all methods of Pasta will be verified before the creation of the first Pasta object, i.e., after printing 1. In C#, where methods are jit-compiled before the first invocation, the method eat need only be verified after printing 3. This is shown in Table 5.

3. Outline of the model

In this section we give an introduction to the model. In the next section we describe the model in full detail. In Fig. 2 we give an overview of our terms and judgements, and the figures where they are defined.

e	expressions	Fig. 3
t	types	Fig. 3
$\imath$	addresses	Fig. 3
$\jmath$	offsets	Fig. 3
`nllPEx`	the null-pointer exception	Fig. 3
`lnkEx`	link-related exception, e.g., verification, load err.	Fig. 3
fa, ma, a	field, method, or any annotation	Fig. 3
$\bar{\kappa}$	field descriptions	Fig. 3
$\bar{\mu}$	method descriptions	Fig. 3
κ	field layout tables	Fig. 3
μ	method layout tables	Fig. 3
v	code tables	Fig. 3
H	heaps	Sec. 4
E	environment giving types to receiver/argument	Sec. 4
$\sqsubset \cdot \sqsupset^{exe}$	execution context	Sect. 4
$\sqsubset \cdot \sqsupset^{off}$	offset calculation context	Sect. 4
$P, H, e \rightsquigarrow_W P', H', e'$	execution in global context W	Fig. 4
$a \rightsquigarrow_P a'$	offset calculation	Fig. 6
$P, e \rightsquigarrow_{W,E} P', e', t$	verification or jit-compilation	Fig. 8
$P, t', t \rightsquigarrow_W P'$	t' is a subtype of t, while extending program P to P'	Fig. 8
$P \rightsquigarrow_W P'$	program P' extends program P in global context W	Fig. 7
$P \vdash t' \leqslant t$	in program P the type t' is a subtype of t	Fig. 5
$\vdash P$	well-formed program	Fig. 9
$P \vdash H$	well-formed heap H for program P	Fig. 10
$P, H \vdash e : t$	runtime expression e has type t in the context of P and H	Fig. 11
$P, H \vdash \imath \lhd c$	$\imath$ conforms class c, or subclass	Fig. 10
$x' \ll x$	x' is more defined that x	Def. 1
$g' \leqslant g$	mapping g' extends g	Def. 1
$g' \leqslant g$ exc. A	mapping g' extends g except in A	Def. 1
$g \otimes g'$	update of g with g', when $\mathcal{D}(g') \subseteq \mathcal{D}(g)$	Def. 1
$g \oplus g'$	update of g with g', when $\mathcal{D}(g') \cap \mathcal{D}(g) = \emptyset$	Def. 1
$g\Downarrow$	extracts pairs corresponding to g	Def. 1
$\mathcal{D}(f), \mathcal{R}(f)$	the domain and range of function f	Def. 1
$FdOffs(P, c)$	the set of all offsets allocated for the fields of c in P	P. 11
$TypFld(P, c, \jmath)$	the type of the field contained at the offset $\jmath$ of c in P	P. 16
$Offst(P, c, t, f)$	the offset of field f as defined c or some superclass	P. 20

Fig. 2. Overview of terms and judgements.

We use the term *raw* class to indicate a class as just loaded, and *laid out* class to indicate a class whose field and method layouts have been determined (the method may, but need not have been verified/jit-compiled). With the concept of programs, P, we describe code in all its forms: raw classes, laid out classes, and method bodies before and after verification/jit-compilation. Programs map identifiers to classes, and addresses to method bodies. Classes contain their superclass names, and are either raw or laid out. Raw classes contain the signatures of fields and methods as well as method bodies; laid out classes contain layout tables, which map field and method signatures to offsets, and virtual method tables, which map offsets to addresses. Global contexts, W, represent the context from which raw classes may be loaded i.e., the file system, or the registry, etc.; therefore, W can be viewed as an abstraction over class loaders, or the versioning system.

Heaps, H, map addresses to objects. Expressions, e, allow for object creation, method invocation, field access and assignment. Execution reads classes from a global context W, and modifies heaps, expressions, and programs. Therefore, it has the format $P, H, e \rightsquigarrow_W P', H', e'$. Loading, verification and laying out of classes can be understood as enriching the information in the program, and is represented through the judgement $P \rightsquigarrow_W P'$. Loading is represented through an extension of P according to the contents of W. The layout tables of a subclass are required to extend those of the superclass. Offset calculation has the format $e \rightsquigarrow_P e'$, meaning that symbolic references in e are replaced by offsets in e', according to the layout tables in P.

Verification/jit-compilation is represented through the judgement $P, e \rightsquigarrow_{W,E} P', e', t$ which means that e is verified/jit-compiled into expression e' with type t. The program P may need to be extended to P', using information from W. The typing needs a typing environment E. Verification may need to check subtypes: $P, t' , t \rightsquigarrow_W P'$ means that t' was established as a subtype of t, and in the process, P was extended to P'.

The model is highly non-deterministic, supporting the description of both Java and C#. In particular, the non-determinism caters for the following four differences:

Offset calculation is "*lazier*'' *in Java*: Verification and jit-compilation are combined into one judgement, $P, e \rightsquigarrow_{W,E} P', e't$. This judgement requires optional offset calculation for its subexpressions (third, fifth and sixth rule in Fig. 8). Optional offset calculation either replaces symbolic references by numeric offsets (first and second rule in Fig. 6), or leaves the symbolic reference unmodified (last rule in Fig. 6). The first alternative describes that C# jit-compilation calculates all offsets. The second alternative describes that Java verification does not calculate any offsets. Furthermore, optional offset calculation may take place during execution (last rule in Fig. 4), and the operational semantics for member access requires the offset to have been calculated (fourth and fifth rules in Fig. 4). This describes the Java "lazy'' offset calculation.
Our model allows many more executions (which do not correspond to either Java or C#), e.g., offsets may be calculated even if not required, and verification/jit-compilation may replace only some of the symbolic references by offsets.

Subtypes are "*optimistic*" *in Java*: Our model considers any class identifier a subtype of itself (last rule in Fig. 8); thus reflecting Java. On the other hand, any class may be loaded during program extension (third rule in Fig. 7), and programs may be extended during verification/jit-compilation (fourth rule in Fig. 8), thus reflecting C#.

Verification is "*lazier*'' *in* C#: The model requires methods to have been verified/jit-compiled before being invoked (fourth rule in Fig. 4), thus describing the C# "lazy" approach. Furthermore, verification/jit-compilation is part of program extension (fifth rule in Fig. 7), and program extension may take place at any time during execution (first rule in Fig. 4), thus describing the Java "eager" approach.
Of course, our model also allows further behaviours, e.g., where only some methods are verified/jit-compiled, or where classes are verified eagerly, upon loading.

Timing and causes of link-related actions: In our model, program extension, which can occur through class loading, verification/jit-compilation, and layout calculation, may take place at any time (first rule in Fig. 4), even if not needed. Furthermore, in our model, a linking exception (not a null pointer exception) may be thrown at any time (second rule in Fig. 4), even if the exception is not necessary. Also, the different kinds of link-related exceptions are not distinguished.

This non-determinism encompasses many execution strategies, including some that are impractical, but simplifies the model considerably.

4. The model

Notation: All mappings are implicitly partial and finite. The terms $\mathcal{D}(g)$, $\mathcal{R}(g)$ denote, respectively, the domain and range of function g. The notation $x' \ll x$ indicates that the values x, x', which may belong to any domain, are equal up to x being ε—in other words, that x' is more defined than x.[3] In order to describe program extension, we define the concepts of mapping extension (i.e., $g' \leqslant g$, and $g' \leqslant g$ exc. A), and the update of g with another mapping g' (i.e., $g \otimes g'$ and $g \oplus g'$). We also define the operation $g\Downarrow$ which extracts all the pairs corresponding to g:

Definition 1. For values x, x' from domain, mappings g, g', and set A

$\mathcal{D}(g)$		the domain of g,
$\mathcal{R}(g)$		the range of g,
$x' \ll x$	iff	$x = \varepsilon$ or $x = x'$,
$g' \leqslant g$	iff	$\mathcal{D}(g) \subseteq \mathcal{D}(g')$, and $g'\vert_{\mathcal{D}(g)} = g$,
$g' \leqslant g$ exc. A	iff	$\mathcal{D}(g') = \mathcal{D}(g) \cup A$, and $g'_{\vert\mathcal{D}(g)\setminus A} = g_{\vert\mathcal{D}(g)\setminus A}$,
$g \cdot g'$	=	a function g'', with $\mathcal{D}(g'') = \mathcal{D}(g) \cup \mathcal{D}(g')$, and $g''(x) = g'(x)$ if $g'(x) \neq \varepsilon$, $g(x)$ otherwise,
$g \otimes g'$	=	$g \cdot g'$ if $\mathcal{D}(g') \subseteq \mathcal{D}(g)$, ε otherwise,
$g \oplus g'$	=	$g \cdot g'$ if $\mathcal{D}(g') \cap \mathcal{D}(g) = \emptyset$, ε otherwise,
$g\Downarrow$	=	$\{\langle x, y\rangle \mid g(x) = y\}$.

Note that the relations $\_ \ll \_$ and $\_ \leqslant \_$ are reflexive, and not symmetric. The operation $\_ \oplus \_$ is commutative, but $\_ \otimes \_$ is not. When $g' \leqslant g$ exc. A holds, the set A and $\mathcal{D}(g)$ may, but need not, be disjoint.

Programs reflect the internal representation of code, and are described in Fig. 3. They map identifiers to raw (*ClassRaw*) or laid out classes (*ClassLaidOut*), and addresses to method bodies. Raw classes correspond to the representations found in `*.class` files (in Java) or `*.dll/*.exe` files (in .NET). They consist of the superclass name, the field descriptors ($\bar{\kappa} \in FldDescr$, consisting of field identifiers and types), and method descriptors ($\bar{\mu} \in MthDescr$, consisting of method identifier, argument type, return type and method body).[4] Laid out classes consist of a field layout table ($\kappa \in FldTbl$, which determines the offset for a field with given identifier and type), the method layout table ($\mu \in MthTbl$, which maps method signatures to offsets), and the virtual table ($v \in CdeTbl$, which maps offsets to addresses of method bodies).[5]

We included class layout and virtual tables in our model, because they are useful to demonstrate what might go wrong if code were able to fool the verifier; execution would not be stuck, instead, any part of the memory could be accessed [11,10,13]. Unverified method bodies consist of a signature and expression, $Typ \times Typ \times Exp$. Verified method bodies consist of an expression, Exp.

Throughout this paper, we extract implicitly components from tuples, e.g., $P(c)$ is a shorthand for $P\downarrow_1(c)$, and $P(\imath)$ is a shorthand for $P\downarrow_2(\imath)$. The notation $P(c) = \langle \_, \_, \_\rangle$ describes that c is still raw, whereas the notation $P(c) = \langle \_, \_, \_, \_\rangle$ indicates that c has been laid out.

Expressions: The syntax of expressions is given in Fig. 3. In expressions we allow imperative features (field assignments), because we believe that they introduce important aspects to the soundness issues relevant for dynamic linking.

Expressions are given in an augmented high level language, near to Java and C# source code. The augmentations are memory offsets and type annotations; both serve to disambiguate field accesses and method invocations (this corresponds to the level of abstraction of Java bytecode and .NET IL). For example, the expression p.cal [Pasta, **int**] denotes the field called cal of type **int**, in the object p, and declared in class Pasta (or superclass). This symbolic reference will be replaced during offset calculation; e.g., if **int** cal has offset 3 in class Pasta, then the expression will be rewritten to p[3].

---

[3] Notice that the notations $x' \ll x$ and $g' \leqslant g$ do *not* follow the usual definition of $\sqsubseteq$, where $\perp \sqsubseteq v$ for any value v. Instead, the notations $x' \ll x$ and $g' \leqslant g$, conform to the notation for subtypes, and express that g' (or x') is more defined than g (or x).

[4] Note that we use the overbar to indicate similar entities from different domains, that is, $\bar{\kappa}$ and $\bar{\mu}$ indicate field and method descriptions in raw classes, while κ and μ indicate field and method layout tables in laid out classes (in particular, overbar is not used to denote vectors).

[5] The Appendix contains an example clarifying descriptions and layout tables in the presence of method inheritance and field hiding.

Programs

$P \in Prg$	$=$	$(ClassId \to (ClassRaw \uplus ClassLaidOut))$	
		$\times (\mathbb{N} \to Body)$	programs
$ClassRaw$	$=$	$ClassId \times FldDescr \times MthDescr$	
$\bar{\kappa} \in FldDescr$	$=$	$FieldId \to Typ$	field descriptions
$\bar{\mu} \in MthDescr$	$=$	$MethId \times Typ \times Typ \to Exp$	method descriptions
$ClassLaidOut$	$=$	$ClassId \times FldTbl \times MthTbl \times CdeTbl$	
$\kappa \in FldTbl$	$=$	$FieldId \times Typ \to \mathbb{N}^+$	field layout tables
$\mu \in MthTbl$	$=$	$MethId \times Typ \times Typ \to \mathbb{N}$	method layout tables
$\nu \in CdeTbl$	$=$	$\mathbb{N} \to \mathbb{N}$	code tables
$Body$	$=$	$(Typ \times Typ \times Exp)$	meth. body before jit/verif.
		$\uplus Exp$	meth. body after jit/verif.

Global contexts

W	$\in$	$ClassId \to ClassRaw$	

Expressions

$e, e' \in Exp$	$::=$	`new` c |	instance creation
		$\imath$ |	address
		`p` |	parameter
		$e\ ma(e')$ |	method invocation
		$e\ fa = e'$ |	field assignment
		$e\ fa$ |	field access
		`this` |	this reference
		`nllPEx` |	null-pointer exception
		`lnkEx`	linking related exception
$t, t' \in Typ$	$::=$	c	type (class name)
$ma \in Ann^M$	$::=$	$.m[c, t, t']$ |	unresolved method annotation
		$[\jmath]$	resolved method annotation
$fa \in Ann^F$	$::=$	$.f[c, t]$ |	unresolved field annotation
		$[\jmath]$	resolved field annotation
$a \in Ann$	$::=$	fa |	field annotation
		ma	method annotation
$c \in ClassId$	$=$	Id	class identifiers
$f \in FieldId$	$=$	Id	field identifiers
$m \in MethId$	$=$	Id	method identifiers
$\imath$	$\in$	$\mathbb{N}$	addresses
$\jmath$	$\in$	$\mathbb{N}$	offsets

Fig. 3. Expressions and programs.

Values are addresses, which are natural numbers denoted by $\imath, \imath'$, etc.; the null pointer is $\mathbf{0}$.[6] `nllPEx` is the exception raised when a field is accessed or a method is invoked on $\mathbf{0}$. Also, `lnkEx` stands for, and does not distinguish between,

[6] Adding further values, e.g., booleans or integers would be possible, but would not add to the description of dynamic linking. In the examples we use more types, e.g., **int** and **String**.

any link related exception, e.g., verification error, class not found error, class circularity error, etc. An expression is *ground*, if it is an address $\imath$ or an exception.

The runtime model: Heaps, H, map addresses to objects, which are blocks of memory consisting of a class identifier and values for the fields. Values are addresses, including **0**. Heaps therefore have the form

$$H : \mathbb{N}^+ \rightarrow \mathbb{N} \biguplus \mathit{ClassId}.$$

We implicitly require the sets $\mathbb{N}$ and *ClassId* to be disjoint. The lookup $H(\imath)$ returns the contents at $\imath$ in H. If $H(\imath) = c \in \mathit{ClassId}$ then $\imath$ points to an object of class c. The fields of that object are stored at some offset, $\jmath$, from $\imath$. An address $\imath$ is *fresh* in heap H iff $\forall \jmath : H(\imath + \jmath) = \varepsilon$ (that is, H is undefined for all addresses greater than or equal to $\imath$).

The following heap, H_0, contains a Penne object at 2, and a Food object at 4:

$H_0(2) =$ Penne start Penne object,
$H_0(3) =$ *55* field **int** cal from Pasta,
$H_0(4) =$ Food start Food object,
$H_0(\imath) = \varepsilon$ for all other $\imath$ s.

Note, that the structure of an object is not reflected in our heap model, e.g., the heap does not describe which fields belong to which object. Thus, as in [13], heaps are modelled at a lower level than in verifier studies [40,27,37], where objects are indivisible entities, and where there are no address calculations. Our lower level model can describe the potential damage when executing unverified code. [7]

Execution modifies the current program, expression and heap. It therefore has the format

$$P, H, e \rightsquigarrow_W P', H', e'.$$

This judgement reflects that execution happens in a global context W, that programs may be extended, expressions get rewritten, and heaps may be modified. The judgement is defined through small step semantics in Fig. 4.

Evaluation is the part of execution not directly affected by dynamic linking. It is described by the third through eighth rule in Fig. 4.

Creation of a new object of class c, through the expression `new` c, allocates fresh addresses for the fields of c at the corresponding offsets, initializing them with **0**. It requires the auxiliary function $\mathit{FdOffs}(P, c)$ which collects the field offsets from all superclasses [8]:

$$\mathit{FdOffs}(P, c) = \bigcup_{P \vdash c \leqslant c'} \mathcal{R}(P(c') \downarrow_2).$$

Method invocation, $\imath[\jmath](\imath')$, looks up the method body e in $H(\imath)$, the dynamic class of the receiver $\imath$, using the offset $v(\jmath)$, and executes that body after replacing `this` by the actual receiver $\imath$, and the parameter `p` by the argument $\imath'$. Therefore, evaluation only applies to expressions which do *not* contain `this`, or `p`. The format of the invocation $\imath[\jmath](\imath')$ (rather than $(\imath.m[c, t_r, t_p](\imath'))$ means that the offset has been calculated. The requirement $P(c) = \langle \_, \_, \_, v \rangle$ (rather than $P(c) = \langle \_, \_, \_ \rangle$) means that the class c has been laid out. The requirement that $P(v(\jmath)) = e$ (rather than $P(v(\jmath)) = \langle \_, \_, \_ \rangle$) means that the particular method has been verified/jit-compiled (Fig. 4).

Field lookup retrieves the contents of the heap at the given offset, whereas field assignment updates the heap at the given offset, as in the fifth rule. Method invocation and field access for **0** throw a `nllPEx`, as described in the sixth rule of the table.

---

[7] On the other hand, our model distinguishes the sets *ClassId* and $\mathbb{N}$, so it contains more information than the plain bitstrings, found in real memory. Faithful modelling of that aspect would not have promoted the study of dynamic linking.

[8] Note that the function $\mathit{FdOffs}(P, c)$ is well-defined, even if the program P should contain cycles in the class hierarchy.

$$\frac{P \rightsquigarrow_W P'}{P, H, e \rightsquigarrow_W P', H, e} \qquad \frac{}{P, H, e \rightsquigarrow_W P, H, \texttt{lnkEx}}$$

$$\frac{FdOffs(P, c) = \{\jmath_1, \ldots, \jmath_n\},\ \imath \text{ fresh in } H}{P, H, \texttt{new}\, c \rightsquigarrow_W P, H[\imath \mapsto c, \imath + \jmath_1 \mapsto \mathbf{0}, \ldots \imath + \jmath_{\mathbf{n}} \mapsto \mathbf{0}], \imath}$$

$$\frac{H(\imath) = c \quad P(c) = \langle \_, \_, \_, v \rangle \quad P(v(\jmath)) = e}{P, H, \imath[\jmath](\imath') \rightsquigarrow_W P, H, e[\imath/\texttt{this}, \imath'/\texttt{p}]} \qquad \frac{\imath \neq \mathbf{0}}{P, H, \imath[\jmath] \rightsquigarrow_W P, H, H(\imath + \jmath)} \quad P, H, \imath[\jmath] = \imath' \rightsquigarrow_W P, H[\imath + \jmath \mapsto \imath'], \imath'$$

$$\frac{}{P, H, \mathbf{0}[\jmath] \rightsquigarrow_W P, H, \texttt{nllPEx}} \quad P, H, \mathbf{0}[\jmath] = \imath \rightsquigarrow_W P, H, \texttt{nllPEx} \quad P, H, \mathbf{0}[\jmath](\imath) \rightsquigarrow_W P, H, \texttt{nllPEx}$$

$$\frac{P, H, e \rightsquigarrow_W P', H', e'}{P, H, \sqsubset e \sqsupset^{exe} \rightsquigarrow_W P', H', \sqsubset e' \sqsupset^{exe}} \qquad \frac{z = \texttt{nllPEx},\ \text{or } z = \texttt{lnkEx}}{P, H, \sqsubset z \sqsupset^{exe} \rightsquigarrow_W P, H, z}$$

$$\frac{a \rightsquigarrow_P a'}{P, H, \sqsubset a \sqsupset^{off} \rightsquigarrow_W P, H, \sqsubset a' \sqsupset^{off}}$$

Fig. 4. Execution and evaluation.

$$\frac{P(c_1)\downarrow_{\mathrm{I}} = c_2}{P \vdash c_1 \leqslant c_1 \quad P \vdash c_1 \leqslant c_2} \qquad \frac{P \vdash c_1 \leqslant c_2 \quad P \vdash c_2 \leqslant c_3}{P \vdash c_1 \leqslant c_3}$$

Fig. 5. Subtypes.

Execution is propagated to its context, as described in the seventh rule. Both link related and link unrelated exceptions (i.e., z) are propagated out of their contexts, as described in the eighth rule. Execution contexts allow a succinct description of propagation

$$\begin{array}{rcl} \sqsubset\cdot\sqsupset^{exe} & ::= & \sqsubset\cdot\sqsupset^{exe} ma(e) \ \mid\ \imath\, ma(\sqsubset\cdot\sqsupset^{exe}) \ \mid \\ & & \sqsubset\cdot\sqsupset^{exe} fa = e \ \mid\ \imath fa = \sqsubset\cdot\sqsupset^{exe} \ \mid\ \sqsubset\cdot\sqsupset^{exe} fa \end{array}$$

Optional offset calculation may replace a symbolic annotation through a numeric offset, and has the format

$$a \rightsquigarrow_P a',$$

where a represents a field or method annotation. The first rule in Fig. 6 says that offsets for fields are looked up in the field layout table of the particular class c, under the given type t, and field identifier f. The second rule in Fig. 6 says that offsets for methods are looked up in the method layout table of the particular class, under the given argument and return types, and method identifier. Thus, a class may inherit or define several methods with the same names and argument type but different result type, and it may inherit fields with same name and types as its own fields.[9] The last

[9] An example of this is shown in the Appendix.

$$\frac{P(c) = \langle \_, \kappa, \_, \_ \rangle \quad \kappa(f,t) = \jmath}{.f[c,t] \rightsquigarrow_P [\jmath]} \qquad \frac{P(c) = \langle \_, \_, \mu, \_ \rangle \quad \mu(m, t_r, t_p) = \jmath}{.m[c, t_r, t_p] \rightsquigarrow_P [\jmath]} \qquad \frac{}{a \rightsquigarrow_P a}$$

Fig. 6. Optional offset calculation.

$$\frac{}{P \rightsquigarrow_W P} \qquad \frac{P'' \rightsquigarrow_W P' \quad P \rightsquigarrow_W P''}{P \rightsquigarrow_W P'} \qquad \frac{W(c) = \langle c_s, \_, \_ \rangle \quad P(c_s) \neq \varepsilon}{P \rightsquigarrow_W P \oplus [c \mapsto W(c)]}$$

$$\frac{\begin{array}{ll} P(c) = \langle c_s, \bar{\kappa}, \bar{\mu} \rangle, \ P(c_s) = \langle \_, \kappa_s, \mu_s, \nu_s \rangle & (a) \\ \kappa \text{ injective}, \ \mathcal{D}(\kappa) = \bar{\kappa}\Downarrow, \ \mathcal{R}(\kappa) \cap \mathit{FdOffs}(P, c_s) = \emptyset & (b) \\ \mu \text{ injective}, \ \mu \leqslant \mu_s, \ \mathcal{D}(\mu) = \mathcal{D}(\mu_s) \cup \mathcal{D}(\bar{\mu}) & (c) \\ \nu \leqslant \nu_s \ \text{ exc. } \mu(\mathcal{D}(\bar{\mu})) & (d) \\ \bar{\mu}(m,t,t') = e \Longrightarrow & (e) \\ \quad \exists j, 1 \leqslant j \leqslant n : \ \imath_j = \nu(\mu(m,t,t')), \ t_j = t, \ t'_j = t', \ e_j = e & \end{array}}{P \rightsquigarrow_W P \otimes [c \mapsto \langle c_s, \kappa, \mu, \nu \rangle] \oplus [\imath_1 \mapsto \langle t_1, t'_1, e_1 \rangle, \ \ldots, \ \imath_n \mapsto \langle t_n, t'_n, e_n \rangle]}$$

$$\frac{\begin{array}{ll} P(c) = \langle \_, \_, \mu, \nu \rangle, \ \imath = \nu(\mu(\_, t_r, t_p)), \ P(\imath) = \langle t_r, t_p, e_0 \rangle & (a) \\ \imath \in \mathcal{R}(P(c')\downarrow_4) \implies P \vdash c' \leqslant c & (b) \\ P, e_0 \rightsquigarrow_{W, \{\mathtt{this} \mapsto c, \mathtt{p} \mapsto t_p\}} P'', e, t & (c) \\ P'', t, t_r \rightsquigarrow_W P' & (d) \end{array}}{P \rightsquigarrow_W P' \otimes [\imath \mapsto e]}$$

Fig. 7. Program extension.

rule allows optional offset calculation to leave a unmodified, and is used to model verification as in Java—this is shown in the example at the end of this section.

The last rule in Fig. 4 allows offset calculation to happen during execution, as in Java. For this, we have defined offset calculation contexts as

$$\sqsubset \cdot \sqsupset^{off} \ ::= e \sqsubset \cdot \sqsupset^{off} \ \mid e \sqsubset \cdot \sqsupset^{off} = e \mid e \sqsubset \cdot \sqsupset^{off} \ (e)$$

Optional offset calculation takes place also during verification/jit-compilation (Fig. 8). If one of the two first rules from Fig. 6 is applied, then we obtain C# jit-verification; if the last rule is applied, then we obtain Java verification.

Program extension: A program P' extends another program P, if P' contains more information (through loading of classes), or more refined information (through verification, jit-compilation or layout calculation) than P. This relationship has the format

$$P \rightsquigarrow_W P',$$

cf. Fig. 7, and is defined in the global context of a W which expresses the environment (possibly a file system) from which classes are loaded. The particular environment is not needed for the proof of soundness—it was omitted e.g., in the model in [13], but is needed when formulating and proving equivalence of strategies. In more detail, $P \rightsquigarrow_W P'$ is

defined as follows:

- The first and second rules state that $P \rightsquigarrow_W P'$, if P' is in the reflexive and transitive closure of the extension relation $P \rightsquigarrow_{\ldots} \ldots$.
- The third rule describes the introduction into P of the raw version of class c, as read from W, provided that its superclass, c_s, is already in P.
- The fourth rule describes laying out class c, where the entry for c is replaced by the laid out version $\langle c_s, \kappa, \mu, v\rangle$, and the unverified method bodies from the raw version of c are given fresh addresses $\imath_1, \ldots, \imath_n$:
  (a) In P the class c is raw, and, c_s, the direct superclass of c, is laid out;
  (b) κ, the field layout of c, is distinct from that of all superclasses, and all fields introduced in the raw version of c get fresh offsets;
  (c) μ, the method layout of c, extends [10] that of c_s (thus all methods in c that override methods in c_s retain their offsets in μ), and all methods introduced in the raw version of c are given offsets;
  (d) v, the virtual method table of c, extends that of c_s except for the methods introduced in class c (thus all methods inherited and not overridden by local methods in c, retain their offsets from v_s);
  (e) each method introduced in the raw version of c is mapped by the virtual method table to a fresh address which contains the method body and signature.

  Note, that we use the operator $\otimes$ to denote that P already contained an entry for c, and the operator $\oplus$ to denote that the addresses $\imath_1, \ldots, \imath_n$ are "fresh'' in P.
- The fifth rule describes the replacement of the unverified method $\langle t_r, t_p, e_0\rangle$ by the verified method body e:
  (a) the new program is the outcome of verification of an unverified method body, found through the method layout and virtual table of a class c at address $\imath$;
  (b) the class c is the most general superclass of all classes c' which may contain the address $\imath$ in their virtual table. Fig. 5 define the subtype relation;
  (c) verification takes place in an environment which considers the receiver to belong to class c and the parameter to have type t_p—as found in the signature from the entry in class c;
  (d) the outcome of verification has a type which is subtype of the one given in the method's signature.

As we said earlier, program extensions may take place at any time during execution (cf. Fig. 4).

Verification and jit-compilation: We describe the similarities between Java verification and C# jit-compilation through the verification/jit-compilation judgement

$$P, e \rightsquigarrow_{W,E} P', e', t$$

defined in Fig. 8, which transforms an expression e to e', type checks e to have type t, and possibly extends the program P to P'. The process takes place in an environment E which maps `this` and the parameter `p` to types, i.e., E: $\{$ `this`, `p` $\} \to Typ$, and in a global context W, from which further raw classes may be loaded. The parameter `p` and the receiver `this` have the type given in the environment E. Verification/jit-compilation of an object creation expression requires c to be a class, and gives it type c. The value **0** has any class type c.

Method invocation requires the receiver and argument to be well-typed, and to be of subtypes of c and t_p, the receiver and argument types stored in the symbolic method annotation $.m[c, t_r, t_p]$. The method invocation has type t_r, the result type of the annotation. The symbolic annotation may be replaced by an offset, thus modelling C# jit-compilation. Offset calculation also allows for the identity, thus modelling Java verification. Similar explanations apply to the rules which access fields.

Finally, verification may require classes to be loaded, and the offset calculation may require layout information about some classes. This is described through the fourth rule, which allows extension of the program at any time—the use of this rule is demonstrated in the example at the end of this section.

Verification/jit-compilation may need to check that a type is a subtype of another type, and while doing so may need to load further classes, as in judgement

$$P, t', t \rightsquigarrow_W P'$$

[10] For simplicity, we do not model the C# `new` modifier that introduces a method with the same signature as an inherited one without overriding it.

$$\frac{}{P, \texttt{this} \rightsquigarrow_{W,E} P, \texttt{this}, E(\texttt{this})} \qquad \frac{P, c, c \rightsquigarrow_W P'}{P, \texttt{new}\ c \rightsquigarrow_{W,E} P', \texttt{new}\ c, c}$$

$$\frac{}{P, \texttt{p} \rightsquigarrow_{W,E} P, \texttt{p}, E(\texttt{p})} \qquad \frac{}{P, \mathbf{0} \rightsquigarrow_{W,E} P', \mathbf{0}, c}$$

$$\frac{\begin{array}{l} P, e_1 \rightsquigarrow_{W,E} P_1, e_1', t_1 \\ P_1, e_2 \rightsquigarrow_{W,E} P_2, e_2', t_2 \\ P_2, t_1, c \rightsquigarrow_W P_3 \\ P_3, t_2, t_{\mathrm{p}} \rightsquigarrow_W P' \\ .m[c, t_{\mathrm{r}}, t_{\mathrm{p}}] \rightsquigarrow_{P'} ma \end{array}}{P, e_1.m[c, t_{\mathrm{r}}, t_{\mathrm{p}}](e_2) \rightsquigarrow_{W,E} P', e_1'\ ma(e_2'), t_{\mathrm{r}}} \qquad \frac{\begin{array}{l} P \rightsquigarrow_W P'' \\ P'', e \rightsquigarrow_{W,E} P', e', t \end{array}}{P, e \rightsquigarrow_{W,E} P', e', t}$$

$$\frac{\begin{array}{l} P, e_1 \rightsquigarrow_{W,E} P_1, e_1', t_1 \\ P_1, e_2 \rightsquigarrow_{W,E} P_2, e_2', t_2 \\ P_2, t_1, c \rightsquigarrow_W P_3 \\ P_3, t_2, t_f \rightsquigarrow_W P' \\ .f[c, t_f] \rightsquigarrow_{P'} fa \end{array}}{P, e_1.f[c, t_f] = e_2 \rightsquigarrow_{W,E} P', e_1'fa = e_2', t_f} \qquad \frac{\begin{array}{l} P, e \rightsquigarrow_{W,E} P_1, e', t_e \\ P_1, t_e, c \rightsquigarrow_W P' \\ .f[c, t_f] \rightsquigarrow_{P'} fa \end{array}}{P, e.f[c, t_f] \rightsquigarrow_{W,E} P', e'fa, t_f}$$

$$\frac{\begin{array}{l} P \rightsquigarrow_W P' \\ P' \vdash t' \leqslant t \end{array}}{P, t', t \rightsquigarrow_W P'} \qquad \frac{}{P, t, t \rightsquigarrow_W P}$$

Fig. 8. Verification and jit-compilation.

which is also given in Fig. 8. The format of this judgement expresses that, in some sense, program P, and types t and t' are "input'', while program P' is "output''. Notice that the last rule in Fig. 8 allows any identifier to be a subtype of itself even if the identifier has not been loaded—this follows the "optimistic'' Java approach.

An example: We demonstrate the interplay of verification, execution and extension in terms of the following example: consider a program P_1 which contains a raw version of class Pasta but does not contain an entry for Penne, an environment E_1 where p is declared to have type Penne, and a global context W_1, where Penne extends Pasta. Consider also P_2, which extends P_1, through the raw version of Penne, and P_3, which extends both P_1 and P_3 through the laid out version of Pasta and the raw version of Penne.

Then, verification of the expression p.cal[Pasta,int] would need to load the class Penne. Thus, through application of the penultimate rule in Fig. 8, and the third rule from Fig. 7, we have P_1, Penne, Pasta$\rightsquigarrow_{W_1} P_2$, thus expressing that the verifier could establish that Penne is a subtype of Pasta, and in the process extended the program to P_2. Then, through application of the sixth rule in Fig. 8, and the third rule in Fig. 6, we obtain P_1, p.cal[Pasta,int]$\rightsquigarrow_{W_1,E_1} P_2$, p.cal[Pasta,int], int. The above reflects verification as in Java.

On the other hand, verification/jit-compilation in C# includes offset calculation. Thus, through application of the fourth and the penultimate rule in Fig. 8, and the first rule in Fig. 6, we obtain P_1, p.cal[Pasta,int]$\rightsquigarrow_{W_1,E_1} P_3$, p.cal[1], int. Finally, in Java, offset calculation is interleaved with execution. Thus, through application of the first rule in Fig. 6, and the first and the last rule in Fig. 4, we obtain from any heap H that P_2, H, p.cal[Pasta,int]$\rightsquigarrow_{W_1} P_3$, H, p.cal[1].

5. Soundness

Well-formed programs: The judgement $\vdash P$, describing well-formed programs, is defined in Fig. 9 and requires the following:

(a) the superclass of any raw class from P is defined in P;

$$P(c) = \langle c_s, \_, \_\rangle \implies P(c_s) \neq \varepsilon \qquad (a)$$

$$P(c) = \langle c_s, \_, \_, \_\rangle \implies P(c_s) = \langle \_, \_, \_, \_\rangle \qquad (b)$$

$$P(c) = \langle \_, \kappa, \mu, v\rangle \implies \begin{cases} \kappa, \mu \text{ injective} \\ \mathcal{R}(\mu) = \mathcal{D}(v) \\ \mathcal{R}(v) \subseteq \mathcal{D}(P{\downarrow_2}) \end{cases} \qquad (c)$$

$$\left.\begin{array}{l} c \neq c_s, \\ P \vdash c \leqslant c_s, \\ P(c) = \langle \_, \kappa, \mu, \_\rangle, \\ P(c_s) = \langle \_, \kappa_s, \mu_s, \_\rangle \end{array}\right\} \implies \begin{cases} \mathcal{R}(\kappa_s) \cap \mathcal{R}(\kappa) = \emptyset \\ \mu \leqslant \mu_s \end{cases} \qquad (d)$$

$$P(c) = \langle \_, \_, \mu, v\rangle, \quad P(v(\mu(\_, t_r, t_p)) = e \implies \exists e_0, t : P, e_0 \rightsquigarrow_{\emptyset, \{\mathtt{this} \mapsto c, \mathtt{p} \mapsto t_p\}} P, e, t, \text{ and } P \vdash t \leqslant t_r \qquad (e)$$

$$\vdash P$$

Fig. 9. Well-formed programs.

(b) the superclass of a laid out class is itself laid out;
(c) for any laid out class, the field and method tables map to distinct offsets, the method table maps onto entries in the code table, and the code table maps onto entries in the program's method bodies;
(d) for a laid out class c with some superclass c_s, the fields declared in c have different offsets than those in c_s, and the methods inherited from c_s preserve their offsets into c;
(e) any method body reachable from a method and code table through a given signature is the result of some jit-compilation/verification, which satisfies that signature.

In contrast to our prior work [18], and in the interest of simplicity, we do not require the existence of a class `Object`, nor the code layout table to be injective, nor the existence of a most common superclass for any code shared among classes, nor the class hierarchy to be acyclic. The absence of cycles in class hierarchies is not required for the proof of soundness of the type system; nevertheless, it *is* required by commercial programming languages, probably because such cycles are actually useless.

Cycles in the class hierarchy of well-formed programs are not only allowed, but, because of requirement (a), at least one such cycle is required, if the domain of P is finite. Because of requirement (d), any classes involved in a cycle are required to have no fields, and have methods for the same set of identifiers and signatures. On the other hand, the program extension rules never create new cycles; in particular, the requirement that the superclass of any newly loaded class must be defined in P, guarantees that the subclass relationship for the classes being loaded forms a tree.

Conformance: Fig. 10 defines conformance. The judgement $P, H \vdash \imath$ expresses that the object stored at $\imath$ conforms to its class, c, as stored in $H(\imath)$. For all fields of c, the object must contain appropriate values at the corresponding offsets, and no other object may be stored between its fields. The type of a field at offset $\jmath$ in a particular class c is described through the auxiliary function *TypFld*$(P, c, \jmath)$ [11]:

$$\mathit{TypFld}(P, c, \jmath) = \begin{cases} \bot & \text{if } \jmath \notin \mathit{FdOffs}(P, c), \\ t & \text{if } P(c){\downarrow_2}(\_, t) = \jmath, \\ \mathit{TypFld}(P, P(c){\downarrow_1}, \jmath) & \text{otherwise.} \end{cases}$$

[11] Note that *TypFld*$(P, c, \jmath)$ is well-defined even if the class hierarchy in P contains cycles.

$$\frac{P \vdash c' \leqslant c \quad H(\imath) = c'}{P, H \vdash \imath \lhd c} \qquad \frac{}{P, H \vdash \mathbf{0} \lhd c}$$

$$\frac{H(\imath) = c \quad \forall \jmath.\ TypFld(P, c, \jmath) = t \Longrightarrow P, H \vdash H(\imath + \jmath) \lhd t \quad 1 \leqslant \jmath \leqslant max(FdOffs(P, c)) \Longrightarrow H(\imath + \jmath) \notin ClassId}{P, H \vdash \imath}$$

$$\frac{H(\imath) \in ClassId \Longrightarrow P, H \vdash \imath}{P \vdash H}$$

Fig. 10. Conformance.

$$\frac{}{P, H \vdash \mathbf{0} : c} \qquad \frac{H(\imath) = c' \quad P \vdash c' \leqslant c}{P, H \vdash \imath : c} \qquad \frac{P, H \vdash e : c' \quad P \vdash c' \leqslant c}{P, H \vdash e.f[c, t] : t}$$

$$\frac{}{P, H \vdash \texttt{new}\ c : c} \qquad \frac{P, H \vdash e : c \quad TypFld(P, c, \jmath) = t}{P, H \vdash e[\jmath] : t} \qquad \frac{P, H \vdash e.fa : t \quad P, H \vdash e' : t' \quad P \vdash t' \leqslant t}{P, H \vdash e.fa = e' : t}$$

$$\frac{P, H \vdash e_1 : c_1 \quad P, H \vdash e_2 : t_2 \quad P \vdash c_1 \leqslant c \quad P \vdash t_2 \leqslant t_{\mathrm{p}}}{P, H \vdash e_1.m[c, t_{\mathrm{r}}, t_{\mathrm{p}}](e_2) : t_{\mathrm{r}}} \qquad \frac{P, H \vdash e_1 : c_1 \quad P, H \vdash e_2 : t_2 \quad P \vdash t_2 \leqslant t_{\mathrm{p}} \quad P(c_1) = \langle \_, \_, \mu, \_ \rangle \quad \mu(m, t_{\mathrm{r}}, t_{\mathrm{p}}) = \jmath}{P, H \vdash e_1[\jmath](e_2) : t_{\mathrm{r}}}$$

Fig. 11. Types of runtime expressions.

The judgement $P \vdash H$ requires all objects to conform to their class, and (implicitly) also requires the class of any objects stored in H to be defined in P. Because $\mathbf{0}$ conforms to any class, an object with a field initialized to $\mathbf{0}$ may conform to a class c, even if c', the class of that field in c, has not been loaded yet.

Types for runtime expressions: Types for runtime expressions are described by the judgement $P, H \vdash e : t$, from Fig. 11, with rules similar to those for verification/jit-compilation, with the difference that heaps *are* taken into account (to give types to addresses), environments are *not* taken into account (runtime expressions do not contain `this`, or `p`), and the program is *not* extended. Runtime expressions containing offsets for method invocation are typed by application of the inverse method layout (in well-formed programs the method layouts are injective, hence their inverses are defined).

Preservation of properties: We prove that verification/jit-compilation and execution extend programs, and that when a program P is extended to P', while trying to establish a subtype relationship, then the subtype relationship holds in P'.

Lemma 1. *For any* $e, e', P, P', P'', H, H', H'', t, t'$:
1. $P, e \rightsquigarrow_{W,E} P', e', t \implies P \rightsquigarrow_W P'$,
2. $P, H, e \rightsquigarrow_W P', H', e' \implies P \rightsquigarrow_W P'$,
3. $P, t, t' \rightsquigarrow_W P' \implies P' \vdash t' \leqslant t$.

Proof Sketch. The proofs follow from the definition of the two rewrite relationships, $\ldots \rightsquigarrow_{W,E} \ldots$ and of $\ldots \rightsquigarrow_W \ldots$. □

We can now prove that if we can verify an expression in an environment where the receiver belongs to a class c, then we can also verify that expression in an environment where the receiver belongs to a subclass of c:

Lemma 2.

$$\left.\begin{array}{l} P, e_0 \rightsquigarrow_{W,\{\mathtt{this}\mapsto c, \mathtt{p}\mapsto t_\mathtt{p}\}} P, e, t \ \textit{and} \\ P \vdash c' \leqslant c \end{array}\right\} \implies \left\{\begin{array}{l} P, e_0 \rightsquigarrow_{W,\{\mathtt{this}\mapsto c', \mathtt{p}\mapsto t_\mathtt{p}\}} P, e, t' \\ P \vdash t' \leqslant t \end{array}\right. .$$

Proof Sketch. By structural induction on $P, e_0 \rightsquigarrow_{W,\{\mathtt{this}\mapsto c, \mathtt{p}\mapsto t_\mathtt{p}\}} P, e, t$. □

Properties such as subtyping, conformance of the heap, runtime type of an expression, verification of an expression, or well-formedness of a program, established in a program P are preserved in an extending program P'. Similar properties were proven in [16], used in [10,13], and explored in our model of binary compatibility [17]. Notice that such properties do not always hold for source code, cf. [6] for counterexamples.

Lemma 3. *If* $P \rightsquigarrow_W P'$, *then*
1. $P'(c)\!\downarrow_1 \ll P(c)\!\downarrow_1$,
2. $\mathit{TypFld}(P', c, \jmath) \ll \mathit{TypFld}(P, c, \jmath)$,
3. $\mathit{FdOffs}(P', c) \ll \mathit{FdOffs}(P, c)$,
4. $P \vdash t_1 \leqslant t_2 \implies P' \vdash t_1 \leqslant t_2$,
5. $P, H \vdash \imath \implies P', H \vdash \imath$,
6. $P \vdash H \implies P' \vdash H$,
7. $P, H \vdash e : t \implies P', H \vdash e : t$,
8. $P, t', t \rightsquigarrow_W P \implies P', t', t \rightsquigarrow_W P'$,
9. $P, e \rightsquigarrow_{W,E} P, e', t \implies P', e \rightsquigarrow_{W,E} P', e', t$.[12]

Proof Sketch. Assertions 1–3 are proven by structural induction over the judgement $P \rightsquigarrow_W P'$. The remaining assertions are proven by structural induction over the judgements of each of the assertions, i.e., over $P \vdash t_1 \leqslant t_2$, or $P, H \vdash \imath$, etc. □

Lemma 4. $P \rightsquigarrow_W P'$ *and* $\vdash P \implies \vdash P'$.

Proof Sketch. By structural induction over the derivation of $P \rightsquigarrow_W P'$, and then a proof that all requirements of $\vdash P'$ are satisfied, applying Lemmas 2 and 3. □

A direct corollary of Lemmas 1 and 4 is that execution of *any* expression preserves well-formedness of programs.

---

[12] Notice that the premise $P, e \rightsquigarrow_{W,E} P, e', t$ does not allow extension of the program P. Although the lemma could be generalized to allow for extensions, the current restricted form suffices for the proof of soundness.

If an expression is the outcome of jit-compilation/verification, then replacement of the receiver and argument by addresses pointing to objects of appropriate classes, preserves its type [13] :

Lemma 5. *For any* $P, e, e', c, t, \imath, \imath', t_{\mathtt{p}}$:

$$\left.\begin{array}{l} \vdash P \\ P, e' \rightsquigarrow_{\emptyset,\{\mathtt{this}\mapsto c,\mathtt{p}\mapsto t_{\mathtt{p}}\}} P, e, t \\ P, H \vdash \imath \lhd c \\ P, H \vdash \imath' \lhd t_{\mathtt{p}} \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \exists\, t' : \\ P, H \vdash e[\imath/\mathtt{this}, \imath'/\mathtt{p}] : t', \\ P \vdash t' \leqslant t. \end{array}\right.$$

Notice that the lemma from above does *not* need to require that the heap is well-formed!

Proof Sketch. Use structural induction over the judgement $P, e' \rightsquigarrow_{\emptyset,\{\mathtt{this}\mapsto c,\mathtt{p}\mapsto t_{\mathtt{p}}\}} P, e, t$. The base cases are straightforward. Let us consider the inductive case where the last rule applied was the sixth rule from Fig. 8. This implies that e' is a field access, i.e., has the form $e_1.f[c, t_f]$. We distinguish two cases: first Case: the optional offset calculation for $.f[c, t]$ replaces the symbolic annotation by a numeric offset, and second Case: the optional offset calculation leaves the symbolic annotation unmodified. In the first Case, we use the fact that $.f[c, t] \rightsquigarrow_P [\jmath]$, and $\vdash P$ imply that $\mathit{TypFld}(P, c, \jmath) = t$; the rest follows by application of the induction hypothesis, and the type rules from Fig. 11. In the second Case, the hypothesis follows directly from application of the induction hypothesis, and the type rules from Fig. 11.

The other inductive cases are analogous. □

Execution of a well-typed expression e does not overwrite objects, rather it creates any new objects in the free space. Also, execution does *not* affect the type of any expression e''—even if e'' were a subexpression of e. This is required for type soundness in imperative object oriented languages, and was proven, e.g., in [16,42,13]. In the current work it holds only for *well-typed* expressions e.

Lemma 6. *If* $P \vdash H$, *and* $\vdash P$, *and* $P, H \vdash e : t$, *and* $P, H, e \rightsquigarrow_W P', H', e'$, *then*
1. $H(\imath) = c \Longrightarrow H'(\imath) = c$,
2. $H'(\imath) = c \Longrightarrow H(\imath) = c$ *or* $\imath$ *free in* H,
3. $P, H \vdash e'' : t'' \Longrightarrow P', H' \vdash e'' : t''$.

Proof Sketch. Assertions 1 and 2 are proven by structural induction over the derivation $P, H, e \rightsquigarrow_W P', H', e'$. The last assertion is proven by structural induction over the typing of e''. The requirements $\vdash P$ and $P, H \vdash e : t$ are needed in order to guarantee that memory is accessed in "appropriate" ways only. Note that such requirements were not needed for the corresponding lemmas for high level description languages e.g., [16]; they are needed here, because we have a lower level model of the heap.

Soundness: Subject reduction guarantees that the heap H' preserves conformance, uninitialized parts of the store are never dereferenced, and the expression preserves its type.

Theorem 1. *For any* e, P, H, H', t

$$\left.\begin{array}{l} P \vdash H \\ \vdash P \\ P, H \vdash e : t \\ P, H, e \rightsquigarrow_W P', H', e' \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} P' \vdash H', \\ \textit{if } e' \textit{ does not contain exceptions, then} \\ \exists t' : \ P', H' \vdash e' : t', \quad P' \vdash t' \leqslant t. \end{array}\right.$$

Proof Sketch. By structural induction over the typing of e. The proof strategy is similar to that used for many imperative, small object oriented languages, e.g., [14]. What is new here, is the fact that the underlying program is extended during execution. This is why we needed to prove that program execution creates programs which extend the original one (Lemmas 1.1 and 1.2), preserves all judgements (Lemma 3), and in particular program well-formedness (Lemma 4).

[13] Notice that if the receiver and argument were not replaced, the expression might not have a runtime type, since runtime types are assigned in the absence of an environment, and since environments are necessary in order to give a type to this and to p.

We now sketch the case where the last rule applied in the typing of e is the last rule from Fig. 11. Then e is a function call. We then proceed by case analysis over the rule applied in the execution of $P, H, e \rightsquigarrow_W P', H', e'$:

First Case: The last rule applied was the first rule from Fig. 4. The lemma follows through Lemmas 3.3, and 3.7, application of the inductive hypothesis, and Lemma 4.
Second Case: The last rule applied was the context rule, i.e., the seventh rule from Fig. 4. The lemma follows through application of the inductive hypothesis, and the typing rules.
Third Case: The last rule applied was the offset calculation rule, i.e., the last rule from Fig. 4, and the second rule from Fig. 6. The lemma follows through application of the last rule in Fig. 11.
Fourth Case: The last rule applied was the fourth rule from Fig. 4, and replaced the method call by the method body.

The lemma follows through application of well-formedness of the program, and Lemma 5. □

Execution is never stuck in that a well-typed expression can always be further reduced. For instance, further classes can always be loaded, and the program extended. However, these reductions are not always interesting, and do not always mean "real progress" of the expression being executed. This is why we do not formulate a progress lemma.

Progress, in the sense of always being able to further reduce a well-typed expression by another well-typed expression does not hold, since we do not have a closed program, and there exists the possibility that a verification error will be thrown, or that a class cannot be loaded, or a resolution error will be thrown by offset calculation. We do not model these errors explicitly; instead, when in a situation where the Java or C# runtime system would, e.g., throw a verification error, in our system, none of the verification rules would be applicable—in fact only program extension and the error rules would be applicable.

6. Equivalence of execution strategies

In this section we show that all execution strategies are equivalent, i.e., that non-determinism does *not* affect the result of evaluations which do not throw link related exceptions. The global context W needs to be explicitly stated here. The theorem does *not* apply for intermediate results, nor if z were a link related exception—several counterexamples were shown in Section 2.

Theorem 2. *For any global context W, and any e, P, P', P'', H, H', H'', $\imath$, and $z, z' \in \mathbb{N} \cup \{\texttt{nllPEx}\}$*

$$\left.\begin{array}{l} P, H, e \rightsquigarrow^*_W P', H', z \\ P, H, e \rightsquigarrow^*_W P'', H'', z' \end{array}\right\} \Longrightarrow z = z', H' = H'' \quad \textit{up to renaming of addresses.}$$

Note that we do not require the programs to be well-formed. Also, we do require that both executions, $P, H, e \rightsquigarrow^*_W P', H', z$, and $P, H, e \rightsquigarrow^*_W P'', H'', z'$ take place in the same global context W.

Proof Sketch. The proof of Theorem 2 is the most demanding of all the proofs in this paper, and requires the introduction of some auxiliary concepts. We will need to clarify the meaning of "up to renaming of addresses", and we will need to tighten the definition of programs.

In addition to the structural requirements for programs, as defined in Fig. 3, we ask that the expressions found in the laid out or raw classes do not contain addresses or non-symbolic annotations (i.e., offsets), and that the virtual method table of a class contains an entry for each entity from the method layout table. These requirements are guaranteed in well-formed programs, but in the current theorem we are not requiring the programs to be well-formed. More formally, for the purposes of this theorem, we require any program P to satisfy

- $P(c) = \langle \_, \_, \mu, v \rangle \Longrightarrow \mathcal{R}(\mu) \subseteq \mathcal{D}(v)$,
- $P(c, m, t_r, t_p) = e \implies e$ does not contain addresses nor non-symbolic annotation.

The function $W(c, m, t_r, t_p)$ looks up the class c in W, and returns the method body for m in class c, with result and parameter types t_r and t_p—if it exists. The function $P(c, m, t_r, t_p)$ returns the method body for m, in class c, with result and parameter types t_r and t_p, respectively, independently of whether the class c has been laid out in P, and whether

the method body has been verified or not.

$$W(c, m, t_r, t_p) = \begin{cases} e & \text{if } W(c) = \langle \_, \_, \bar{\mu} \rangle, \bar{\mu}(m, t_r, t_p) = e, \\ \varepsilon & \text{otherwise,} \end{cases}$$

$$P(c, m, t_r, t_p) = \begin{cases} e & \text{if } P(c) = \langle \_, \_, \bar{\mu} \rangle, \bar{\mu}(m, t_r, t_p) = e, \\ e & \text{if } P(c) = \langle \_, \_, \mu, \nu \rangle, P(\nu(\mu(m, t_r, t_p))) = \langle t_r, t_p, e \rangle, \\ e & \text{if } P(c) = \langle \_, \_, \mu, \nu \rangle, P(\nu(\mu(m, t_r, t_p))) = e, \\ \varepsilon & \text{otherwise.} \end{cases}$$

It is easy to show that the functions $P(c, m, t_r, t_p)$ and $W(c, m, t_r, t_p)$ are well-defined, i.e., that exactly one of the cases from above will hold.

We now define the auxiliary function $\mathit{Offst}(P, c, t, f)$ which returns the offset of field f of type t as defined in class c or some superclass:

$$\mathit{Offst}(P, c, t, f) = \begin{cases} \varepsilon & \text{if } P(c) = \langle \_, \_, \_ \rangle, \text{ or} \\ & \quad P(c) = \langle \_, \_, \_, \_ \rangle, \text{ and} \\ & \quad \forall c', P \vdash c \leqslant c', P(c) = \langle \_, \kappa, \_, \_ \rangle : \kappa(f, t) = \varepsilon, \\ \jmath & \text{if } P(c) = \langle \_, \kappa, \_, \_ \rangle, \kappa(f, t) = \jmath, \\ \jmath & \text{otherwise, and where } \jmath = \mathit{Offst}(P, P(c)\downarrow_1, t, f). \end{cases}$$

Note that $\mathit{Offst}(P, c, t, f)$ as defined above, is well-defined, even if the class hierarchy in P should contain cycles.

In order to define "up to renaming of addresses'', we use the concept of a *heap renaming*, a bijective mapping

$$\sigma : \mathbb{N} \to \mathbb{N} \text{ where } \sigma(\mathbf{0}) = \mathbf{0},$$

which renames addresses across two heaps, preserving the address $\mathbf{0}$. Using σ, in the next paragraph we will define relations across heaps, expressions, and programs. First, we give their intuitive meaning in this paragraph:

1. $\sigma \vdash z \sim z'$ means that z and z' are equivalent addresses or they are both the null pointer exception.
2. $P, P', H, H', \sigma \vdash \imath \sim \imath'$ means that addresses $\imath$ and $\imath'$ point to equivalent objects, i.e., to objects of the same class, and whose fields can be found at equivalent addresses.
3. $P, P', \sigma \vdash H \sim H'$ means that heaps H and H' are equivalent, in the sense that the heap renaming function maps objects onto equivalent objects.
4. $P, P', c, t_f \vdash fa \sim fa'$ means that the field annotations fa and fa' are equivalent in the sense that they are either both symbolic and identical, or, if they are offsets, then these offsets correspond to looking up a field of type t_f from a class c.
5. $P, P', c, t_r, t_p \vdash ma \sim ma'$ means that the method annotations ma and ma' are equivalent in the sense that they are either both symbolic and identical, or, if they are offsets, then these offsets correspond to looking up a method with parameter type t_p, return type t_r, from a class c.
6. $P, P', H, H', E, \sigma \vdash e \sim e' : t$ means that the expressions e and e' are equivalent, i.e., that they have the same structure up to the replacement of addresses, and corresponding offsets, and can be considered to have type t. [14]
7. $P, P', E \vdash e' \sim e$ is the counterpart to $P, P', H, H', E, \sigma \vdash e \sim e' : t$ for expressions which do not contain addresses, and where the ensuing type does not matter.
8. $\vdash P \sim P'$ means that programs P and P' are equivalent in the sense that field and method layout tables are equivalent, and that if there are entries for method bodies in *both* programs, then they contain equivalent expressions.
9. $\sigma \vdash P, H \sim P', H'$ means that H and H' are equivalent and P and P' are equivalent.
10. $W \vdash P$ expresses that the contents of the program P "agree'' with those in the global environment W.

We now formally define the equivalence relationships

1. $\sigma \vdash z \sim z'$ iff
   (a) $z \equiv \texttt{nllPEx} \equiv z'$, or
   (b) $z, z' \in \mathbb{N}$, and $\sigma(z) = z'$.

[14] We are using the vague term "can be considered to have type'' to express that the expressions are not necessarily well-typed in the sense of Fig. 11; the judgement $P, P', H, H', E, \sigma \vdash e \sim e' : t$ does not check that subexpressions "fit'' their environment, e.g., for field assignment we do not require the right-hand side to be a subtype of the left-hand side.

2. $P,\ P', H, H', \sigma \vdash \imath \sim \imath'$ iff $\exists c$, such that
   (a) $H(\imath) = c$, and $H'(\imath') = c$,
   (b) $P(c) = \langle \_, \_, \_, \_ \rangle,\ P'(c) = \langle \_, \_, \_, \_ \rangle$,
   (c) $\forall f, t_f :\ \mathit{Offst}(P, c, t_f, f) \neq \varepsilon$ if and only if $\mathit{Offst}(P', c, t_f, f) \neq \varepsilon$,
   (d) $\forall f, t_f, \jmath, \jmath' : \jmath \neq \varepsilon$ and $\jmath = \mathit{Offst}(P, c, t_f, f)$ and $\jmath' = \mathit{Offst}(P', c, t_f, f \Longrightarrow) \sigma(\imath + \jmath) = \imath' + \jmath'$.
3. $P,\ P', \sigma \vdash H \sim H'$ iff $\sigma(\imath) = \imath' \implies P,\ P', H, H', \sigma \vdash \imath \sim \imath'$.
4. $P,\ P', c, t_f \vdash fa \sim fa'$ iff one of the following cases holds:
   (a) $fa \equiv fa',\ fa \equiv \_[\_, \_]$, [15]
   (b) $fa \equiv .f[c, t_f], fa' \equiv [\jmath]$, and $P'(c)\downarrow_2 (f, t_f) = \jmath$,
   (c) $fa \equiv [\jmath], fa' \equiv .f[c, t_f]$, and $P(c)\downarrow_2 (f, t_f) = \jmath$, [16]
   (d) $fa \equiv [\jmath], fa' \equiv [\jmath']$, and $\exists f : P(c)\downarrow_2 (f, t_f) = \jmath, P'(c)\downarrow_2 (f, t_f) = \jmath'$.
5. $P,\ P', c, t_\mathrm{r}, t_\mathrm{p} \vdash ma \sim ma'$ iff one of the following cases holds:
   (a) $ma \equiv ma', ma \equiv \_[\_, \_, \_]$, [17]
   (b) $ma \equiv .m[c, t_\mathrm{r}, t_\mathrm{p}], ma' \equiv [\jmath]$, and $P'(c)\downarrow_3 (m, t_\mathrm{r}, t_\mathrm{p}) = \jmath$,
   (c) $ma \equiv [\jmath], ma' \equiv .m[c, t_\mathrm{r}, t_\mathrm{p}]$, and $P(c)\downarrow_3 (m, t_\mathrm{r}, t_\mathrm{p}) = \jmath$, [18]
   (d) $ma \equiv [\jmath], ma' \equiv [\jmath'], \exists m,\ P(c)\downarrow_3 (m, t_\mathrm{r}, t_\mathrm{p}) = \jmath,\ P'(c)\downarrow_3 (m, t_\mathrm{r}, t_\mathrm{p}) = \jmath'$.
6. $P,\ P', H, H', E, \sigma \vdash e' \sim e : t$ iff one of the following cases holds:
   (a) $e \equiv \mathtt{this} \equiv e'$, and $t = E(\mathtt{this})$,
   (b) $e \equiv \imath,$ and $e' \equiv \imath'$, and $\sigma(\imath) = \imath'$, and $t = H(\imath)$,
   (c) $e \equiv \mathtt{p} \equiv e'$, and $t = E(\mathtt{p})$,
   (d) $e \equiv \mathtt{new}\, c \equiv e'$, and $t \equiv c$,
   (e) $e \equiv e_1 fa$, and $e' \equiv e'_1 fa'$, and $\exists c$ with
      (i) $P,\ P', H, H', E, \sigma \vdash e_1 \sim e'_1 : c$,
      (ii) $P,\ P', c, t \vdash fa \sim fa'$,
   (f) $e \equiv (e_1 fa = e_2)$, [19] and $e' \equiv (e'_1 fa' = e'_2)$, and $\exists c, t'$ with
      (i) $P,\ P', H, H', E, \sigma \vdash e_1 \sim e'_1 : c$,
      (ii) $P,\ P', c, t' \vdash fa \sim fa'$,
      (iii) $P,\ P', H, H', E, \sigma \vdash e_2 \sim e'_2 : t$,
   (g) $e \equiv e_1 ma(e_2)$, and $e' \equiv e'_1 ma'(e'_2)$, and $\exists c, t'$ with
      (i) $P,\ P', H, H', E, \sigma \vdash e_1 \sim e'_1 : c$,
      (ii) $P,\ P', c, t, t_\mathrm{p} \vdash ma \sim ma'$,
      (iii) $P,\ P', H, H', E, \sigma \vdash e_2 \sim e'_2 : t'$.
7. $P,\ P',\ E \vdash e \sim e'$ iff e and e' do not contain addresses, and $e \equiv e'$ or $P,\ P', H, H', E, \sigma \vdash e' \sim e : t$, for some type t, renaming function σ, and heaps H and H'. [20]
8. $\vdash P \sim P'$ iff the following conditions hold:
   (a) $P(c) = \langle \_, \_, \_ \rangle, P'(c) = \langle \_, \_, \_ \rangle \Longrightarrow P(c) = P'(c)$,
   (b) $P(c) = \langle c', \bar{\kappa}, \bar{\mu} \rangle,\ P'(c) = \langle c'', \kappa, \mu, \nu \rangle \Longrightarrow$
      (i) $c' = c''$,
      (ii) $\mathcal{D}(\kappa) = \bar{\kappa}\Downarrow$,
      (iii) $\mathcal{D}(\mu) = \mathcal{D}(\bar{\mu}) \cup \mathcal{D}(P'(c'')\downarrow_3)$,
   (c) $P'(c) = \langle c', \bar{\kappa}, \bar{\mu} \rangle,\ P(c) = \langle c'', \kappa, \mu, \nu \rangle \implies$ ... dual of earlier case,

---

[15] Thus, identical *unresolved* field annotations are equivalent regardless of the particular class c, and type t_f.

[16] This case is the dual of the previous one.

[17] As for field annotations, identical *unresolved* method annotations, are equivalent in the context of *any* class c, result type t_r, and parameter type t_p.

[18] This case is the dual of the previous.

[19] These parantheses are only a means to enhance the readability of the equivalences.

[20] Since the expressions e' and e do not contain addresses, satisfaction of the expression equivalence condition is independent of heaps. More formally, one can show that if e and e' do not contain addresses, then $P,\ P', H, H', E, \sigma \vdash e \sim e' : t$ implies $P,\ P', H'', H''', E, \sigma' \vdash e \sim e' : t$ for *any* H'', H''', and σ'.

(d) $P(c) = \langle c'', \kappa, \mu, v\rangle,\ P'(c) = \langle c''', \kappa', \mu', v'\rangle \implies$
  (i) $c'' = c'''$,
  (ii) $\mathcal{D}(\kappa) = \mathcal{D}(\kappa')$,
  (iii) $\mathcal{D}(\mu) = \mathcal{D}(\mu')$,
(e) $P(c, m, t_r, t_p) = e$, and $P'(c, m, t_r, t_p) = e' \implies P,\ P', E \vdash e \sim e'$ for $E \equiv (\texttt{this} \mapsto c, \texttt{p} \mapsto t_p)$.

9. $\sigma \vdash P, H \sim P', H'$ iff the following conditions hold:
   (a) $P,\ P', \sigma \vdash H \sim H'$,
   (b) $\vdash P \sim P'$.
10. $W \vdash P$, iff
   (a) $P(c) = \langle \_, \_, \_\rangle \implies W(c) = P(c)$,
   (b) $P(c) = \langle c', \kappa, \mu, v\rangle \implies W(c) = \langle c', \bar{\kappa}, \bar{\mu}\rangle$, and
     (i) $\mathcal{D}(\kappa) = \bar{\kappa}\Downarrow$,
     (ii) $\mathcal{D}(\mu) = \mathcal{D}(\bar{\mu}) \cup \mathcal{D}(P(c')\downarrow_3)$,
   (c) $P(c, m, t_r, t_p) = e \implies \exists e'$ so that $W(c, m, t_r, t_p) = e'$ and $e \equiv e'$ or $P,\ P, E \vdash e \sim e'$ for $E \equiv (\texttt{this} \mapsto c, \texttt{p} \mapsto t_p)$.

We describe executions which only employ program extension steps through the relation

$$P, H, e \overset{ext}{\rightsquigarrow}{}^*_W P', H, e,$$

where the notation $\overset{ext}{\rightsquigarrow}{}^*_W$ indicates that execution consists exclusively of program extension steps, which have been applied in the global context W.

We describe executions which do not employ program extension steps through the relation

$$P, H, e \overset{cor}{\rightsquigarrow}{}^* P, H', e',$$

where the notation $\overset{cor}{\rightsquigarrow}{}^*$ indicates that only "core", i.e., non-program extension steps have been applied.

We can show that a core evaluation step followed by a program extension evaluation step can be reversed and give the same effect:

$$(Prop\_1a)\quad \left.\begin{array}{l} P, H, e \overset{cor}{\rightsquigarrow} P, H', e', \\ P, H', e' \overset{ext}{\rightsquigarrow}_W P', H', e'. \end{array}\right\} \implies \left\{\begin{array}{l} P, H, e \overset{ext}{\rightsquigarrow}_W P', H, e, \\ P', H, e \overset{cor}{\rightsquigarrow} P', H', e'. \end{array}\right.$$

We can then show that any evaluation can be broken into two parts, so that all the program extension steps take place first, and the core steps take place after there are no more extension steps, i.e.,

$$(Prop\_1)\ \ P, H, e \rightsquigarrow^*_W P', H', e' \implies \left\{\begin{array}{l} P, H, e \overset{ext}{\rightsquigarrow}{}^*_W P', H, e, \\ P', H, e \overset{cor}{\rightsquigarrow}{}^* P', H', e'. \end{array}\right.$$

We will first study the properties of extension steps. First, we can show that optional offset calculation creates equivalent annotations, i.e.,

$$(Prop\_2a)\quad \begin{array}{ll} .f[c, t_f] \rightsquigarrow {}_P fa & \implies P,\ P, c, t_f \vdash .f[c, t_f] \sim fa, \\ .m[c, t_r, t_p] \rightsquigarrow {}_P ma & \implies P,\ P, c, t_r, t_p \vdash .m[c, t_r, t_p] \sim ma. \end{array}$$

We can also prove that program extension preserves agreement of expressions, i.e.,

$$(Prop\_2b)\quad \left.\begin{array}{l} P,\ P', H, H', E, \sigma \vdash e \sim e' : t, \\ P \rightsquigarrow_W P'' \end{array}\right\} \implies P'',\ P', H, H', E, \sigma \vdash e \sim e' : t.$$

Using $(Prop\_2a)$ and $(Prop\_2b)$ we can prove that jit-compilation/verification creates an equivalent expression, i.e.,

$$(Prop\_2c)\ \ P, e \rightsquigarrow_{W,E} P', e', t \implies P,\ P', E \vdash e \sim e'.$$

Using $(Prop\_2b)$, $(Prop\_2c)$, and structural induction on $P \rightsquigarrow_W P'$, we then prove that program extension preserves agreement with the global context, and creates an equivalent program, i.e.,

$$(Prop\_2d) \quad \left.\begin{array}{l} W \vdash P, \\ P \rightsquigarrow_W P' \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} W \vdash P', \\ \vdash P \sim P'. \end{array}\right.$$

We can also prove that program equivalence is transitive, in the context of the *same* global context W, i.e.

$$(Prop\_2e) \quad \left.\begin{array}{l} W \vdash P,\ W \vdash P',\ W \vdash P'', \\ \vdash P \sim P'',\ \vdash P'' \sim P' \end{array}\right\} \Longrightarrow \vdash P \sim P'.$$

Using the above, we can prove that two evaluations that involve extension steps only, when applied to equivalent programs lead to equivalent programs, and that agreement with the global context is preserved, i.e.,

$$(Prop\_2) \quad \left.\begin{array}{l} W \vdash P, \quad W \vdash P', \\ \vdash P \sim P', \\ P, H, e \overset{ext}{\rightsquigarrow}{}^*_W P'', H, e, \\ P', H', e' \overset{ext}{\rightsquigarrow}{}^*_W P''', H', e' \end{array}\right\} \Longrightarrow \vdash P'' \sim P'''.$$

We now study the properties of core steps. We first show that a single core step preserves equivalence of expressions, and heaps, i.e.,

$$(Prop\_3a) \quad \left.\begin{array}{l} \sigma \vdash P, H \sim P', H', \\ P,\ P', H, H', E, \sigma \vdash e \sim e' : t, \\ P, H, e \overset{cor}{\rightsquigarrow} P, H'', e'', \\ P', H', e' \overset{cor}{\rightsquigarrow} P', H''', e''' \\ e'' \not\equiv \texttt{lnkEx} \not\equiv e''' \end{array}\right\}$$

$$\Longrightarrow \left\{\begin{array}{ll} \exists\, \sigma', \sigma' \leqslant \sigma : & \\ \sigma' \vdash P, H \sim P', H'', & \\ (P,\ P', H'', H''', E, \sigma' \vdash e'' \sim e' : t & \text{or} \\ P,\ P', H'', H''', E, \sigma' \vdash e''' \sim e : t & \text{or} \\ P,\ P', H'', H''', E, \sigma' \vdash e''' \sim e'' : t). & \end{array}\right.$$

Note that $(Prop\_3a)$ allows for three possibilities: the two new expressions may be equivalent (if both steps are offset calculations, [21] or both are *not* offset calculations) or one of the new expressions is equivalent to one of the old ones (if one step is an offset calculation, [22] and the other is not an offset calculation). The proof of $(Prop\_3a)$ is by induction on the depth of the execution of the expression (notice that the context rules allow depth more than one).

We illustrate $(Prop\_3a)$ through a simple example: consider programs P_1, P_2, heaps H_1, H_2, expressions $e_1 \equiv \imath_1.f[c_r, t], e_2 \equiv \imath_2.f[c_r, t]$, and a bijection σ, so that (1) $\sigma \vdash P_1, H_1 \sim P_2, H_2$, and (2) $P_1,\ P_2, H_1, H_2, E, \sigma \vdash e_1 \sim e_2 : t$. Because of (2) we have $\sigma(\imath_1) = \imath_2$. If e_1 evaluates through a core step, then it produces some $e_3 \equiv \imath_1.[\jmath]$, where $\jmath$ is the offset of field f with type t in class c_r in program P. Similarly, core execution of e_2 produces some $e_4 \equiv \imath_2.[\jmath']$, where $\jmath'$ is the offset of field f with type t in class c_r in program P'. Therefore, $P_1,\ P_2, H_1, H_2, E, \sigma \vdash e_3 \sim e_4 : t$.

Furthermore, we also have $P_1,\ P_2, H_1, H_2, E, \sigma \vdash e_3 \sim e_2 : t$. Core execution of e_3 produces a value e_5, which is the contents of the object at $\imath$ in heap H at offset $\jmath$, whereas core execution of e_2 produced e_4. Thus, $P_1, P_2, H_1, H_2, E, \sigma \vdash e_3 \sim e_4 : t$, so that one of the new expressions is equivalent with one of the old ones.

Then, using $(Prop\_3a)$ and induction on the maximal length of the executions, we show that given equivalent configurations (i.e., expressions, programs and heaps equivalent in terms of the same rename functions), different terminating executions which do not involve program extension steps create equivalent programs, heaps and results,

[21] Or they are propagations of offset calculation to the context.

[22] Or it is a propagation of an offset calculation to the context.

i.e.,

$$(Prop\_3)\quad \left.\begin{array}{l}\sigma \vdash P, H \sim P', H',\\ P,\ P', H, H', E, \sigma \vdash e \sim e' : t,\\ P, H, e \overset{cgs}{\rightsquigarrow}{}^{*} P, H'', z,\\ P', H', e' \overset{cgs}{\rightsquigarrow}{}^{*} P', H''', z'\end{array}\right\} \Longrightarrow \left\{\begin{array}{l}\exists\, \sigma',\ \sigma' \leqslant \sigma :\\ \sigma' \vdash P'', H'' \sim P''', H''',\\ \sigma' \vdash z \sim z'.\end{array}\right.$$

We illustrate $(Prop\_3)$ by continuing our earlier simple example. Remember that $P_1,\ P_2, H_1, H_2, E, \sigma \vdash e_3 \sim e_2 : t$. Core execution of e_3 produces in one step a value z_1. Core execution of e_2 requires two steps (offset calculation, and field access), and produces a value z_2. Because $\sigma \vdash P_1, H_1 \sim P_2, H_2$, and $\sigma(\imath_1) = \imath_2$, we also have that $\sigma_2 \vdash z_1 \sim z_2$.

Now, we can formulate our theorem in a precise way as follows:

$$(Thm\_2)\quad \left.\begin{array}{l}W \vdash P,\\ P,\ H,\ e \rightsquigarrow^{*}_{W} P',\ H',\ z,\\ P,\ H,\ e \rightsquigarrow^{*}_{W} P'',\ H'',\ z',\end{array}\right\} \Longrightarrow \left\{\begin{array}{l}\exists\, \sigma :\\ \sigma \vdash P', H' \sim P'', H'',\\ \sigma \vdash z \sim z'.\end{array}\right.$$

(Thm_2) is a consequence of properties $(Prop\_1)$, $(Prop\_2)$, and $(Prop\_3)$. □

Finally, we can prove that environments which are identical in the parts required for execution, can lead to identical results.

Theorem 3 (*Monotonicity of Execution with respect to global contexts*). *For any* e, P, P', H, H', *and* $z \in \mathbb{N} \cup \{\texttt{nllPEx}\}$:

$$\begin{array}{l}P,\ H,\ e \rightsquigarrow^{*}_{W} P',\ H',\ z\\ W\ |_{def(P')} = W'\ |_{def(P')}\end{array} \Longrightarrow P,\ H,\ e \rightsquigarrow^{*}_{W'} P',\ H',\ z.$$

We could probably have replaced the requirement $W\ |_{def(P')} = W'\ |_{def(P')}$ by some weaker requirement which would say that only the parts required by the execution of the expression need to be identical.

7. Conclusions, related and further work

Dynamic linking is a very powerful language feature with complex semantics, and it needs to be well understood. We consider our model to be simple, in view of the complexity of the feature, and also compared to an earlier model for Java [13]. We have achieved simplicity through many iterations over the design, and through the choice of appropriate abstractions:

- we do not distinguish the causes of link-related exceptions;
- we allow link-related exceptions to be thrown at *any* time of execution, even when there exist other legal evaluations;
- we do not prescribe at which point of execution the program will be extended, and so allow "unnecessary'' loading, verification or jit-compilations;
- we combine both loaded and verified code in the single concept of a program; [23]
- we represent programs through mappings rather than texts or data structures.

Most of these abstractions were introduced primarily in order to allow the model to serve for both Java and for C#, but they turned out also to simplify significantly the model.

Non-determinism seems to have been in the Java designers' minds: the specification [29, Section 12.1.1], requires resolution errors to be thrown only when linking actions related to the error are required, but does not state anything about when they are to be discovered. Through non-determinism we distilled the main ingredients of dynamic linking from both languages. We prove type soundness, thus obtaining type soundness both for the Java and the C# strategies, and showed that different strategies within the model do not differ widely.

Extensive literature is devoted to the Java verifier [40,26]. Dynamic loading in Java is formalized in [32], while problems with security in the presence of multiple loaders are reported in [39], a solution presented in [33], which is found flawed and improved upon in [37]. Computation does not preserve types but is type sound. Java's multiple loaders

[23] In [13] we distinguished between these, thus keeping a natural distinction, but having a more complex model.

are modelled in [47] which also shows an intermediate solution between the rigid approach based on the classpath and that which allows arbitrary user-defined loaders. Type safety for a substantial subset of the .NET IL is proven in [28].

Interest in linking as part of the program lifecycle was kindled through [9]. A collection of examples that demonstrate small details of the dynamic linking process in Java can be found in [15]. Separate compilation for Java is discussed in [3]. Module interconnection languages, and mixins [45,4,23,19,24] give explicit control of program composition at source code level.

A scheme for delaying the choice of component to be dynamically linked is introduced in [7]; this flexibility can be achieved by adding type variables to the bytecode, which then get substituted at runtime. The scheme has been implemented on .NET [8]. A computational interpretation for Hilbert's choice operator is suggested in [1]; thus giving a typed foundation for dynamic linking. Types may be replaced by other types during computation, causing global changes of types, but in a type safe manner.

Dynamic linking gave rise to the concept of binary compatible changes, [25,34, Section 13], i.e., changes which do not introduce more linking errors than the code being replaced; the concept is explored in [17,46]. Tools that load the most recent binary compatible version of code were developed for Java [38,5] and C# [20,21]. Current JVMs go even further, and support replacing a class by a class of the same signature, as a "fix-and-continue'' feature [12].

Dynamic software updating [31] supports type safe dynamic reloading of code whose type may have changed, while the system is running. Proteus [41] allows on-line evolution to match source-code evolution and supports runtime updates to functions and types (even while they are executing) in a type-safe and representation-consistent manner. Fortress [2] provides program constructs to enable the programmer to explicitly control linking and program evolution.

Further work includes a better understanding of binary compatible library developments, extension of the model to also allow verification by posting constraints which have to be satisfied upon class loading, as suggested in [37], or to allow field lookup to examine the tables of superclasses as in some of the JVMs, the incorporation of C# assemblies and modules, extensions of the model so as to avoid unnecessary linking steps, and "concretization'' of the model so as to obtain Java or C# behaviour.

Acknowledgements

This work was partly supported by three European Commission FET projects: DART, MOBIUS, and AETHER. We are indebted to Vladimir Jurisic for the many discussions, and the information which first attracted our interests in the area. We are grateful to Davide Ancona for his suggestions to consider the question of equivalence of executions, and to Elena Zucca for many good suggestions for simplification of the formal system. We also thank Christopher Anderson, Alex Buckley, and Mark Skipper for valuable feedback. We are also grateful to the ESOP and the TCS reviewers for many useful comments, and suggestions that helped us improve the explanations.

Appendix. Example showing program extension, description and layout tables

The following example aims to demonstrate some fine points about method and field tables. The source code is given in Fig. 12.

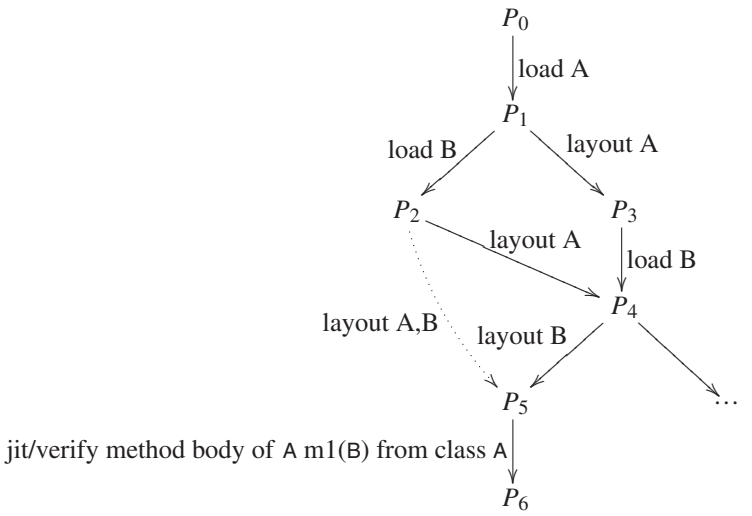
We have two classes, A and B, where A has three fields, f1, f2 and f3. Class B hides f1 with a field of the same type and f2 with a field of a different type—as we shall see the types of the hidden fields do not affect their treatment; also,

```
class A {
    A f1;
    A f2;
    C f3;
    A m1(B p) { e1 }
    B m1(A p) { e2 }
}
```

```
class B extends A {
    A f1;
    B f2;
    B f4;
    A m1(B p) { e3 }
    B m2(B p) { e4 }
}
```

Fig. 12. Example program demonstrating layout.

B introduces a further field f4. Class A introduces the overloaded method m1: there are two versions, one with argument type B, and one with argument type A. The method m1 with argument type B is overridden in B; B also introduces a further method m2.



W

$\mathsf{A} \mapsto \langle \mathsf{Object}, \bar{\kappa}_1, \bar{\mu}_1 \rangle$
$\mathsf{B} \mapsto \langle \mathsf{A}, \bar{\kappa}_2, \bar{\mu}_2 \rangle$

$\bar{\kappa}_1$	$\mathsf{f1} \mapsto \mathsf{A}$
	$\mathsf{f2} \mapsto \mathsf{A}$
	$\mathsf{f3} \mapsto \mathsf{C}$
$\bar{\kappa}_2$	$\mathsf{f1} \mapsto \mathsf{A}$
	$\mathsf{f2} \mapsto \mathsf{B}$
	$\mathsf{f4} \mapsto \mathsf{B}$

$\bar{\mu}_1$	$\langle \mathsf{m1}, \mathsf{A}, \mathsf{B} \rangle \mapsto \mathsf{e}_1$
	$\langle \mathsf{m1}, \mathsf{B}, \mathsf{A} \rangle \mapsto \mathsf{e}_2$
$\bar{\mu}_2$	$\langle \mathsf{m1}, \mathsf{A}, \mathsf{B} \rangle \mapsto \mathsf{e}_3$
	$\langle \mathsf{m2}, \mathsf{B}, \mathsf{B} \rangle \mapsto \mathsf{e}_4$

P_1	A	$\mapsto \langle \mathsf{Object}, \bar{\kappa}_1, \bar{\mu}_1 \rangle$
P_2	A	$\mapsto \langle \mathsf{Object}, \bar{\kappa}_1, \bar{\mu}_1 \rangle$
	B	$\mapsto \langle \mathsf{A}, \bar{\kappa}_2, \bar{\mu}_2 \rangle$
P_3	A	$\mapsto \langle \mathsf{Object}, \kappa_1, \mu_1, v_1 \rangle$
	100	$\mapsto \langle \mathsf{A}, \mathsf{B}, \mathsf{e}_1 \rangle$
	101	$\mapsto \langle \mathsf{B}, \mathsf{A}, \mathsf{e}_2 \rangle$
P_4	A	$\mapsto \langle \mathsf{Object}, \kappa_1, \mu_1, v_1 \rangle$
	B	$\mapsto \langle \mathsf{A}, \bar{\kappa}_2, \bar{\mu}_2 \rangle$
	100	$\mapsto \langle \mathsf{A}, \mathsf{B}, \mathsf{e}_1 \rangle$
	101	$\mapsto \langle \mathsf{B}, \mathsf{A}, \mathsf{e}_2 \rangle$
P_5	A	$\mapsto \langle \mathsf{Object}, \kappa_1, \mu_1, v_1 \rangle$
	B	$\mapsto \langle \mathsf{A}, \kappa_2, \mu_2, v_2 \rangle$
	100	$\mapsto \langle \mathsf{A}, \mathsf{B}, \mathsf{e}_1 \rangle$
	101	$\mapsto \langle \mathsf{B}, \mathsf{A}, \mathsf{e}_2 \rangle$
	102	$\mapsto \langle \mathsf{A}, \mathsf{B}, \mathsf{e}_3 \rangle$
	103	$\mapsto \langle \mathsf{B}, \mathsf{B}, \mathsf{e}_4 \rangle$
P_6	A	$\mapsto \langle \mathsf{Object}, \kappa_1, \mu_1, v_1 \rangle$
	B	$\mapsto \langle \mathsf{A}, \kappa_2, \mu_2, v_2 \rangle$
	100	$\mapsto \mathsf{e}'_1$
	101	$\mapsto \langle \mathsf{B}, \mathsf{A}, \mathsf{e}_2 \rangle$
	102	$\mapsto \langle \mathsf{A}, \mathsf{B}, \mathsf{e}_3 \rangle$
	103	$\mapsto \langle \mathsf{B}, \mathsf{B}, \mathsf{e}_4 \rangle$

κ_1	$\langle \mathsf{f1}, \mathsf{A} \rangle$	$\mapsto 1$
	$\langle \mathsf{f2}, \mathsf{A} \rangle$	$\mapsto 2$
	$\langle \mathsf{f3}, \mathsf{C} \rangle$	$\mapsto 3$
κ_2	$\langle \mathsf{f1}, \mathsf{A} \rangle$	$\mapsto 4$
	$\langle \mathsf{f2}, \mathsf{B} \rangle$	$\mapsto 5$
	$\langle \mathsf{f4}, \mathsf{B} \rangle$	$\mapsto 6$

μ_1	$\langle \mathsf{m1}, \mathsf{A}, \mathsf{B} \rangle$	$\mapsto 0$
	$\langle \mathsf{m1}, \mathsf{B}, \mathsf{A} \rangle$	$\mapsto 1$
μ_2	$\langle \mathsf{m1}, \mathsf{A}, \mathsf{B} \rangle$	$\mapsto 0$
	$\langle \mathsf{m1}, \mathsf{B}, \mathsf{A} \rangle$	$\mapsto 1$
	$\langle \mathsf{m2}, \mathsf{B}, \mathsf{B} \rangle$	$\mapsto 2$

v_1	0	$\mapsto 100$
	1	$\mapsto 101$
v_2	0	$\mapsto 102$
	1	$\mapsto 101$
	2	$\mapsto 103$

Fig. 13. Example demonstrating table layout.

Fig. 13 shows a global context W which describes these classes. Also, it shows a possible sequence of programs involved in execution, and the contents of these programs.

We start with a program P_0, where A and B have not yet been read in—obviously, P_0 contains Object, but we do not show this for the sake of brevity.

Then, we load A, and obtain P_1, for which $P_0 \rightsquigarrow_W P_1$ holds.

From P_1, by loading B, we obtain $P2$, whereas, if we lay out A, we obtain P_3. Therefore, we have $P_1 \rightsquigarrow_W P_2$ and $P_1 \rightsquigarrow_W P_3$ but $W \not\vdash P_2 \leqslant P_3$ and $W \not\vdash P_3 \leqslant P_2$.

We then have $P_3 \rightsquigarrow_W P_4$ through loading of B, and $P_4 \rightsquigarrow_W P_5$ through laying out of class B.

Finally, from P_5 we obtain P_6 jit/verifying the method body m1 of class A located at address 100. Thus, we have that $P_5, \mathsf{e}_1 \rightsquigarrow_{W,\mathtt{this}\mapsto \mathsf{A},\mathtt{p}\mapsto \mathsf{B}} P'', \mathsf{e}'_1, t$ and $P'', t, \mathsf{A} \rightsquigarrow_W P_6$. So, we also have that $P_5 \rightsquigarrow_W P_6$.

References

[1] M. Abadi, G. Gonthier, B. Werner, Choice in dynamic linking, in: Proc. Foundations Software Science and Computation Structures, Springer, Berlin, April 2004.

[2] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, S. Tobin-Hochstadt, The Fortress Language Specification, v0. 707, Sun Microsystems, Inc., July 2005.

[3] D. Ancona, G. Lagorio, E. Zucca, A formal framework for Java separate compilation, in: ECOOP 2002, June 2002.

[4] D. Ancona, E. Zucca, A calculus of module systems, J. Funct. Programming 12 (3) (2002) 91–132.

[5] M. Barr, S. Eisenbach, Safe upgrading without restarting, in: IEEE Conf. Software Maintenance ICSM'2003, IEEE, September 2003.

[6] G. Bracha, Packages break strong locality, Private Communication, February 1999, more at ⟨http://www-dse.doc.ic.ac.uk/sue/packages.html⟩.

[7] A. Buckley, S. Drossopoulou, Flexible dynamic linking, in: Sixth Workshop on Formal Techniques for Java-like Programs, June 2004.

[8] A. Buckley, M. Murray, S. Eisenbach, S. Drossopoulou, Flexible bytecode for linking in .net, in: ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2005), April 2005.

[9] L. Cardelli, Program fragments, linking, and modularization, in: POPL'97 Proc., January 1997.

[10] D. Dean, The security of static typing with dynamic linking, in: Fourth ACM Conf. Computer and Communication Security, 1997.

[11] D. Dean, E.W. Felten, D.S. Wallach, Java security: from HotJava to Netscape and beyond, in: Proc. 1996 IEEE Symp. Security and Privacy, May 1996, pp. 190–200.

[12] M. Dimitriev, Hotspot technology application for advanced profiling, in: ECOOP USE Workshop, June 2002.

[13] S. Drossopoulou, An abstract model of Java dynamic linking and loading, in: R. Harper (Ed.), Types in Compilation, Third International Workshop, Revised Selected Papers, Springer, Berlin, 2001.

[14] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, P. Giannini, Fickle$_{II}$, more dynamic object re-classification, Trans. Programming Languages Systems 24 (2) (2002) 153–191.

[15] S. Drossopoulou, S. Eisenbach, Manifestations of Java dynamic linking—an approximate understanding at source language level, in: The First Workshop on Unanticipated Software Evolution USE'2002, ⟨http://joint.org/use2002/proceedings.html⟩, June 2002.

[16] S. Drossopoulou, S. Eisenbach, S. Khurshid, Is Java Sound?, Theory Practice Object Systems 5 (1) (1999).

[17] S. Drossopoulou, S. Eisenbach, D. Wragg, A fragment calculus—towards a model of separate compilation, linking and binary compatibility, in: LICS Proc., 1999.

[18] S. Drossopoulou, G. Lagorio, S. Eisenbach, Flexible models for dynamic linking, in: Proc. European Symp. Programming, Springer, Berlin, April 2003.

[19] D. Duggan, Sharing in typed module assembly language, in: Preliminary Proc. Third Workshop on Types in Compilation (TIC 2000), Carnegie Mellon, CMU-CS-00-161, 2000.

[20] S. Eisenbach, V. Jurisic, C. Sadler, Managing the evolution of. NET programs, in: Sixth IFIP Internat. Conf. Formal Methods for Open Object-based Distributed Systems, FMOODS'2003, Lecture Notes in Computer Science, Vol. 2884, Springer, Berlin, November 2003, pp. 185–198.

[21] S. Eisenbach, D. Kayhan, C. Sadler, Keeping control of reuseable components, in: IEEE Working Conf. Component Deployment, June 2004.

[22] G. Fenton, E. Felton, Securing Java Getting Down to Business with Mobile Code, Wiley, New York, 1999.

[23] K. Fisher, J. Reppy, J. Riecke, A calculus for compiling and linking classes, in: ESOP Proc., March 2000.

[24] M. Flatt, S. Khrishnamurthi, M. Felleisen, Classes and mixins, in: POPL Proc., January 1998.

[25] I. Forman, M. Conner, S. Danforth, L. Raper, Release-to-release binary compatibility in SOM, in: OOPSLA Proc., October 1995.

[26] S.N. Freund, J.C. Mitchell, A type system for object initialization in the Java bytecode language, in: OOPSLA Proc., October 1998.

[27] S.N. Freund, J.C. Mitchell, A formal framework for the Java bytecode language and verifier, in: OOPSLA Proc., November 1999.

[28] A. Gordon, D. Syme, Typing a multi-language intermediate code, in: Principles of Programming Languages 2001, ACM Press, New York, 2001, pp. 248–260.

[29] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison-Wesley, Reading, MA, August 1996.

[30] A. Hejlsberg, P. Golde, S. Wiltamuth, C# Language Specification, Addison-Wesley, Reading, MA, October 2003.

[31] M. Hicks, J.T. Moore, S. Nettles, Dynamic software updating, in: Programming Language Design and Implementation, ACM, New York, 2001.

[32] T. Jansen, D.L. Metyayer, T. Thorn, A formalization of visibility and dynamic loading in Java, in: IEEE ICCL, 1998.

[33] S. Liang, G. Bracha, Dynamic class loading in the Java™ virtual machine. in: OOPSLA Proc., October 1998.

[34] T. Lindholm, F. Yellin, The Java Virtual Machine, Addison-Wesley, Reading, MA, 1999.

[35] M. Pietrek, Avoiding DLL Hell: introducing application etadata in the Microsoft. NET Framework, in: MSDN Magazine, ⟨msdn.microsoft.com/⟩, October 2000.
[36] S. Pratschner, Simplifying deployment and solving DLL hell with the .NET Framework, ⟨msdn.microsoft.com/⟩, November 2001.
[37] Z. Qian, A. Goldberg, A. Coglio, A formal specification of Java™ class loading, in: OOPSLA'2000, November 2000.
[38] C. Sadler, S. Eisenbach, S. Shaikh, Evolution of distributed Java programs, in: IEEE Working Conf. Component Deployment, June 2002.
[39] V. Saraswat, Java is not type-safe, Technical Report, AT&T Research, 1997, ⟨http://www.research.att.comp/∼vj/bug.html⟩.
[40] R. Stata, M. Abadi, A type system for Java bytecode subroutines, in: POPL'98 Proc., January 1998.
[41] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, I. Neamtiu, Mutatis mutandis: safe and flexible dynamic software updating, in: Proc. ACM Conf. Principles Programming Languages (POPL), January 2005.
[42] D. Syme, Proving Java type sound, in: J. Alves-Foss (Ed.), Formal Syntax and Semantics of Java, Lecture Notes in Computer Science, Vol. 1523, Springer, Berlin, 1999.
[43] Type Safety and Security, in: MSDN Magazine, 2001.
[44] T. Van Vleck (Ed.), Multics Home, ⟨www.multicians.org/⟩, August 2002.
[45] J. Wells, R. Vestergaard, Confluent equational reasoning for linking with first-class primitive modules, in: ESOP Proc., March 2000.
[46] D. Wragg, S. Drossopoulou, S. Eisenbach, What is Java binary compatibility? in: Proc. OOPSLA, Vol. 33, 1998, pp. 341–358.
[47] E. Zucca, S. Fagorzi, D. Ancona, Modeling multiple class loaders by a calculus for dynamic linking, in: Proc. 2004 ACM Symp. Applied Computing, ACM Press, New York, 2004.