Flexible Models for Dynamic Linking

Sophia Drossopoulou[1], Giovanni Lagorio[2], Susan Eisenbach[1] ⋆

[1] Department of Computing at Imperial College, [2] DISI at the University of Genova

Abstract Dynamic linking supports flexible code deployment: partially linked code links further code on the fly, as needed; and thus, end-users receive updates automatically. On the down side, each program run may link different versions of the same code, possibly causing subtle errors which 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 a different order.
We develop a non-deterministic model, which includes the behaviour of Java and C#. 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 in the sense that all terminating executions which do not involve a link error, give the same result.

1 Introduction

Dynamic linking supports flexible code deployment and update: instead of fully linking code before execution, further 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. Dynamic linking was incorporated into operating systems, *e.g.,* Multics, Unix, and Windows, enabling applications to share code, thus saving disk and memory usage. Recently, Java and C# incorporated dynamic linking into the language.

One question connected to dynamic linking is the choice of components to be linked, when there are more than one with the same name. DLLs and .NET offer sophisticated systems of versioning, side-by-side components, registries, *etc*. Difficulties in managing DLLs led to the term "DLL Hell" [19]. The .NET architecture, with assemblies carrying versioning information claims to have solved this problem [20]. Java, on the other hand, links the first class with given name found in the classpath, and any more sophisticated scheme can be implemented through custom class loaders [17].

Another 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 [7,18]. DLLs do not attempt to guarantee type safety: type errors may occur and go undetected, or throw exceptions of an unrelated nature in unrelated parts of the code. Conversely, in Java and C# if the components linked turn out to be "incompatible", link related exceptions are thrown, describing the nature of the problem.

⋆ work partly supported by DART, EU project IST-2001-33477

Thus, although Java and C# do not guarantee choice of compatible components, they guarantee type safety and give error messages that signal the source of the problem.

Our study is concerned with how Java and C# guarantee type safety. Java and C# dynamic linking 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 give the same result. However, Java and C# dynamic linking are not identical: The linking phases have different granularity, are combined differently and take place in a different order. Linking errors may be detected at different times in Java and C# executions.

We develop a non-deterministic model, to describe the behaviour of both Java and C#. We prove soundness, and that all executions that 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 IL. It abstracts from Java 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.

Section 2 introduces Java and C# dynamic linking with an example. Section 3 describes the model. Section 4 states properties, soundness, and equivalence. Section 5 concludes. At `www.disi.unige.it/person/LagorioG/dart/papers/DLE02-long.ps` there is a longer version containing more examples, lemmas, and detail.

2 Introduction to the Dynamic Linking Phases

In the presence of dynamic linking, execution can be understood in terms of;

- evaluation, which is not affected by dynamic linking
- loading, which reads classes from the environment
- verification, which checks type-safety of the code
- laying out, which determines object layout and method tables,
- offset calculation, which replaces references to fields and methods through the corresponding offsets.

These phases apply to different units of granularity: Loading and laying out apply to classes, whilst verification applies to method bodies, and offset calculation applies to individual member access expressions.

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 requires some class to extend a further class it will load the two classes – although [21] suggest a lazier approach of posting constraints instead.

The phases are organized slightly differently in Java than in C#: In Java, offset calculation takes place per instruction, and only before the particular member is accessed, whereas in C#, offset calculation takes place per method, and is combined with verification, to give jit-compilation. In Java, all methods of a class are verified together, whereas

Java	C#	output
	calc. offset for main	
verify Food $\hookrightarrow$ verify main $\hookrightarrow$check Meal $\leq$ Meal $\hookrightarrow$check Penne $\leq$ Penne	jit main $\hookrightarrow$ check Meal $\leq$ Meal $\hookrightarrow$load Meal $\hookrightarrow$ lay out Meal $\hookrightarrow$ check Penne $\leq$ Penne $\hookrightarrow$load Penne; Pasta $\hookrightarrow$`LoadErr` if $\neg$ Cls $\hookrightarrow$ lay out Penne $\hookrightarrow$lay out Pasta $\hookrightarrow$ calc. offset for eat (Penne)	
calc. offset for main		
execute main	execute main	
		—1—
lay out Meal verify Meal $\hookrightarrow$ verify eat (Penne) $\hookrightarrow$ check Penne $\leq$ Pasta $\hookrightarrow$load Penne; Pasta $\hookrightarrow$`LoadErr` if $\neg$ Cls $\hookrightarrow$`VerifErr`, if $\neg$ Sub $\hookrightarrow$ verify chew (Pasta)		
create a new Meal object	create a new Meal object	
		—2—
verify Penne $\hookrightarrow$... $\hookrightarrow$verify Pasta $\hookrightarrow$...		
create a new Penne object	create a new Penne object	
		—3—
calc. offset for eat (Penne)	jit eat (Penne) $\hookrightarrow$check Penne $\leq$ Pasta $\hookrightarrow$`VerifErr`, if $\neg$ Sub $\hookrightarrow$calc. offset for chew (Pasta)	
execute eat (Penne)	execute eat (Penne)	
calc. offset for chew (Pasta)	jit chew (Pasta) $\hookrightarrow$calc. offset for **int** cal from Pasta $\hookrightarrow$`NoFieldErr`, if $\neg$ Fld	
execute chew (Pasta)	execute chew (Pasta)	
		0
		—4—
execute eat (Penne)	execute eat (Penne)	
execute chew (Pasta)	execute chew (Pasta)	
calc. offset for **int** cal from Pasta $\hookrightarrow$`NoFieldErr`, if $\neg$ Fld		
		100

Table 1. Execution of the program example – with verification

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); }
	}

Table 2. Example program

in C# methods are jit-compiled only before execution. The example from table 2 illustrates these points in both Java and C#, (details `www-dse.doc.ic.ac.uk/~sue/foodexample.html`) and consists of classes Meal and Food, compiled in an environment containing compiled versions of Pasta and Penne:

class Pasta { **int** cal = 100; } **class** Penne **extends** Pasta { }

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**

which are required by main in Food, *e.g.,* Sub guarantees successful verification of the eat method body, and Fld guarantees successful field access. If Cls, Sub and Fld hold, execution will be successful, and Java and C# will give the same output.

However, the versions of Pasta and Penne available at runtime might differ from those above: Pasta or Penne may not be available, *i.e.,* $\neg$ Cls. Penne may not be a subtype of Pasta. *i.e.,* $\neg$ Sub. Pasta may not contain a field **int** cal, *i.e.,* $\neg$ Fld.

These situations will lead to linking errors, detected by the corresponding linking phases. Because these take place at different times in Java and C#, the errors will be reported at different times. This is shown in table 1. The third column contains the output, *e.g.,* — 1 —. The first and second column contain the linking phases as they occur in Java or in C#, with their dependencies indicated through the $\hookrightarrow$ symbol, *e.g.,* in Java, verification of class Meal requires verification of method eat, which in its turn checks that Pasta $\leq$ Pasta, and Penne $\leq$ Penne.

The table shows execution both when Cls, Sub, and Fld hold, and when they do not. Thus, if Cls, Sub, and Fld hold, the two executions will print the same output. However:

Verification is "lazier" in C#: Thus, $\neg$ Sub would cause a linking error after —1— in Java, and after —3— in C#. Java verification checks all methods of that class, whereas C# verifies each method when jit-compiling it before its first call.

Offset calculation is "lazier" in Java: Thus, $\neg$ Fld would cause a linking error after —3— in C#, and after —4— in Java. References to fields (or methods) are resolved in Java only when the field is actually accessed during execution, whereas in C# references are resolved when the method containing the reference is jit-compiled.

Subtypes are "optimistic" in Java. Thus, $\neg$ Cls could cause a linking error before —1— in C#, but only after —1— in Java. Checking that a class is a subclass of itself causes loading of the class in C#, but does not in Java.

3 The Model

The appendix lists all the judgments and terms of this model, and their place of definition. All mappings are partial; $dom(f)$, $rng(f)$ denote the domain and range of function f respectively, and ϵ denotes the undefined value.

3.1 Outline of the model Programs, P (see fig. 1), describe code in all its forms, *i.e.,* the "raw" classes as loaded, the method bodies before and after verification/jit-compilation, and the class layout. Ps map identifiers to classes, and addresses to method bodies. Classes contain their superclass names, and they are either "raw", containing the signatures of fields and methods, and method bodies; or, they are "laid out", containing 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.

Heaps, H, map addresses to objects. Expressions, e, allow for method call, field access and assignment. Execution reads classes from a global context W, and modifies heaps, expressions, and programs. Therefore, it has the form: $P, H, e \leadsto_W P', H', e'$.

Loading, verification and laying out of classes can be understood as enriching the information in the program, represented through the judgement $W \vdash P' \leq P$. Loading is represented through an extension of P using the contents of W. The layout tables are required to extend those of the superclass. Verification and jit-compilation is represented through modification of method bodies indicating that they have been verified, and possible substitutions of symbolic references by offsets.

Offset calculation has the format $e \leadsto_P e'$, 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: $P, e \leadsto_{W,E} P', e', t$ which means that e is verified/jit-compiled into expression e' and has 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: $P, t', t \leadsto_W P'$ 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 languages:

Verification is "lazier" in C#. The model requires methods to have been verified/jit-compiled before being called (fourth rule in fig. 3), thus allowing the C# lazy approach. However, verification is part of program extension (fifth rule in fig. 2), and program extension may take place at any time during execution (first rule in fig. 3), thus allowing the Java approach too. Of course, it also allows further behaviour, *e.g.,* where only some methods are verified/jit-compiled, or where classes are verified upon loading.

Offset calculation is "lazier" in Java. The model combines verification and jit-compilation into one judgment, $P, e \leadsto_{W,E} P', e', t$, which requires offset calculation for its subexpressions (third to fifth rules in fig. 5). This describes C# jit-compilation. Offset calculation may also leave the expression unmodified (last rule in fig. 4), and that describes Java verification. Offset calculation may also take place during execution (last rule in fig. 3), and the operational semantics for member access requires the offset to have been calculated (fourth and fifth rules in fig. 3). This describes Java offset cal-

		Programs	
$P \in Prg$	$=$	$(ClassId \to (ClassRaw \uplus ClassLaidOut))$	
		$\times\ (\mathbb{N} \to Body)$	programs
$ClassRaw$	$=$	$ClassId \times \Delta^F \times \Delta^M$	
$\delta^F \in \Delta^F$	$=$	$FieldId \to Typ$	field descriptions
$\delta^M \in \Delta^M$	$=$	$MethId \times Typ \times Typ \to Exp$	method descriptions
$ClassLaidOut$	$=$	$ClassId \times \mathcal{T}^F \times \mathcal{T}^M \times \mathcal{T}^C$	
$\tau^F \in \mathcal{T}^F$	$=$	$FieldId \times Typ \to \mathbb{N}^+$	field layout tables
$\tau^M \in \mathcal{T}^M$	$=$	$MethId \times Typ \times Typ \to \mathbb{N}$	method layout tables
$\tau^C \in \mathcal{T}^C$	$=$	$\mathbb{N} \to \mathbb{N}$	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	
$e, e' \in Exp$	$::=$	$\mathtt{new}\ c \mid$	instance creation
		$\iota \mid$	address
		$\mathtt{y} \mid$	parameter
		$e\ ma(e') \mid$	method invocation
		$e\ fa = e' \mid$	field assignment
		$e\ fa \mid$	field access
		$\mathtt{this} \mid$	this reference
		$\mathtt{nllPExc} \mid$	null-pointer exception
		$\mathtt{lnkExc}$	linking related exception
$t, t' \in Typ$	$::=$	c	type (class name)
$ma \in Ann^M$	$::=$	$.m[c, t, t'] \mid$	unresolved method annotation
		$[\kappa]$	resolved method annotation
$fa \in Ann^F$	$::=$	$.f[c, t] \mid$	unresolved field annotation
		$[\kappa]$	resolved field annotation
$a \in Ann$	$::=$	$fa \mid$	field annotation
		ma	method annotation
$c \in ClassId$	$=$	Id	class identifiers
$f \in FieldId$	$=$	Id	field identifiers
$m \in MethId$	$=$	Id	method identifiers
ι	$\in$	$\mathbb{N}$	addresses
κ	$\in$	$\mathbb{N}$	offsets

Subtypes

$$\frac{P(c_1)\downarrow_1 = c_2}{P \vdash c_1 \le c_1} \qquad \frac{\begin{array}{c} P \vdash c_1 \le c_2 \\ P \vdash c_2 \le c_3 \end{array}}{P \vdash c_1 \le c_3}$$

$$P \vdash c_1 \le c_2$$

Figure 1. Expressions, programs, subtypes

$$\frac{}{W \vdash P \leq P} \qquad \frac{\begin{array}{l} W \vdash P' \leq P'' \\ W \vdash P'' \leq P \end{array}}{W \vdash P' \leq P}$$

$$\frac{\begin{array}{l} P =c= P' \\ P(c) = \epsilon \\ P'(c) = W(c) = \langle c_s, \_, \_ \rangle \\ P(c_s) \neq \epsilon \end{array}}{W \vdash P' \leq P} \qquad \frac{\begin{array}{l} P =c= P' \\ P(c) = \langle c_s, \delta^F, \delta^M \rangle,\ P'(c) = \langle c_s, \tau^F, \tau^M, \tau^C \rangle \\ P(c_s) = \langle \_, \tau_s^F, \tau_s^M, \tau_s^C \rangle \\ \tau^F \text{ injective},\ dom(\tau^F) = \{\langle f, t \rangle \mid \delta^F(f) = t\} \\ rng(\tau^F) \cap FdOffs(P, c_s) = \emptyset \\ \tau^M \leq \tau_s^M \text{ wrt } dom(\delta^M), \quad \tau^C \preceq \tau_s^C \text{ wrt } \tau^M(dom(\delta^M)) \\ \delta^M(m, t, t') = e \implies \quad \exists \iota : \\ \qquad \tau^C(\tau^M(m, t, t')) = \iota, P(\iota) = \epsilon,\ P'(\iota) = (t, t', e) \end{array}}{W \vdash P' \leq P}$$

$$\frac{\begin{array}{l} P =\iota= P' \\ P(\iota) = \langle t_r, t_p, e \rangle, \quad P'(\iota) = e' \\ \exists c \in dom(P') : \\ \qquad P(c') = \langle \_, \_, \_, \tau_1^C \rangle,\ \iota \in rng(\tau_1^C) \implies P \vdash c' \leq c \\ \qquad P', e \leadsto_{W, \{\mathtt{this} \mapsto c, \mathtt{y} \mapsto t_p\}} P', e', t \\ \qquad P' \vdash t \leq t_r \end{array}}{W \vdash P' \leq P}$$

Figure 2. Program extension

culation. The model allows many more executions, *e.g.,* offsets may be calculated even if not required, or *only some* of the symbolic references are replaced.

Subtypes are "optimistic" in Java. The model considers any class identifier a subtype of itself (last rule in fig. 5); thus reflecting Java. However, subtype checking may extend a program during verification (penultimate rule in fig. 5), thus reflecting C#.

Timing of link-related actions The model allows loading, jit-compilation, verification, and offset calculation to take place at any time (first rule in fig. 3), even if not needed. It allows linking exceptions (not null pointer exceptions) at any time (second rule in fig. 3), even if not necessary, and does not distinguish the reason. This does not reflect practical implementations but simplifies the model considerably.

3.2 Programs reflect the internal representation of code. They are described in figure 1. They map identifiers to raw ($ClassRaw$) or to laid out classes ($ClassLaidOut$), and addresses to method bodies. Raw classes correspond to *.class or *.dll files. They consist of the superclass name, the field descriptions ($\delta^F \in \Delta^F$) consisting of field identifiers and types, and method descriptions ($\delta^M \in \Delta^M$) consisting of method identifier, argument type, return type and method body. Laid out classes consist of a field layout table ($\tau^F \in \mathcal{T}^F$), which determines the offset for a field with given identifier and type, the method layout table ($\tau^M \in \mathcal{T}^M$), which maps method signatures to offsets, and the virtual table ($\tau^C \in \mathcal{T}^C$), which maps offsets to addresses of method bodies.

$$\frac{W \vdash P' \leq P}{P, H, e \leadsto_W P', H, e} \qquad \frac{}{P, H, e \leadsto_W P, H, \texttt{lnkExc}}$$

$$\frac{\mathit{FdOffs}(P, c) = \{\kappa_1, \ldots, \kappa_n\}, \quad \iota \text{ free in } H}{P, H, \texttt{new}\ c \leadsto_W P, H[\iota \mapsto c, \iota{+}\kappa_1 \mapsto \mathbf{0}, \ldots \iota{+}\kappa_\mathbf{n} \mapsto \mathbf{0}], \iota}$$

$$\frac{\begin{array}{l} H(\iota) = c \\ P(c) = \langle \_, \_, \_, \tau^C \rangle \\ P(\tau^C(\kappa)) = e \end{array}}{P, H, \iota[\kappa](\iota') \leadsto_W P, H, e[\iota/\texttt{this}, \iota'/\texttt{y}]} \qquad \frac{\iota \neq \mathbf{0}}{\begin{array}{l} P, H, \iota[\kappa] \leadsto_W P, H, H(\iota + \kappa) \\ P, H, \iota[\kappa] = \iota' \leadsto_W P, H[\iota + \kappa \mapsto \iota'], \iota' \end{array}}$$

$$\frac{}{\begin{array}{l} P, H, \mathbf{0}[\kappa] \leadsto_W P, H, \texttt{nllPExc} \\ P, H, \mathbf{0}[\kappa] = \iota \leadsto_W P, H, \texttt{nllPExc} \\ P, H, \mathbf{0}[\kappa](\iota) \leadsto_W P, H, \texttt{nllPExc} \end{array}} \qquad \frac{P, H, e \leadsto_W P', H', e'}{P, H, \sqsubset e \sqsupset^{exe} \leadsto_W P', H', \sqsubset e' \sqsupset^{exe}}$$

$$\frac{z = \texttt{nllPExc}, \text{ or } z = \texttt{lnkExc}}{P, H, \sqsubset z \sqsupset^{exe} \leadsto_W P', H', z}$$

$$\frac{a \leadsto_P a'}{P, H, \sqsubset a \sqsupset^{\mathit{off}} \leadsto_W P, H, \sqsubset a' \sqsupset^{\mathit{off}}}$$

Figure 3. Execution.

Method bodies which have not been checked consist of a signature and expression, $\mathit{Typ} \times \mathit{Typ} \times \mathit{Exp}$. Bodies which have been checked consist of an expression, Exp.

3.3 Expressions The syntax is given in figure 1. It describes classes, subclasses, methods and fields for an imperative language.

We use an augmented high level language, near to source code. The augmentations are memory offsets, and type annotations, used to disambiguate fields or methods with the same name. For example, the expression p.cal [Pasta,**int**] denotes the field called cal of p, of type **int**, and declared in class Pasta. 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].

Values are addresses, natural numbers denoted by ι, ι' *etc*; the null pointer is $\mathbf{0}$. When a field is accessed or a method is called on $\mathbf{0}$, the `nllPExc` exception is raised. Also, `lnkExc` stands for, and does not distinguish between, any link related exception, *i.e.,* verification errors, class not found, class circularities, absence of fields and methods.

3.4 Execution modifies the current program, expression and heap, and has the form

$$P, H, e \leadsto_W P', H', e'$$

expressing that the global context W may be used for program extension. It is defined through small step semantics in figure 3.

Heaps, H, map addresses to objects, which are memory blocks consisting of class identifier, and values for the fields. Values are object addresses, or **0**. Heaps have form:

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

If $H(\iota) = c \in \mathit{ClassId}$ then ι points to an object of class c. The fields of that object are stored at some offset, κ, from ι. An address ι is fresh in H iff $\forall\kappa : H(\iota+\kappa) = \epsilon$.

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	$H_0(\iota)$	= ϵ	for all other ι 's

Thus, as in [4], heaps are modelled at a lower level than in verifier studies [24,10,21], where objects are indivisible entities, and where there are no address calculations. Our lower level model enables the description of the potential damage when executing unverified code.

3.5 Program Extension We define mapping extensions ($g' \leq g$ wrt A, $g' \preceq g$ wrt A), and program equality up to class or address ($P =c= P'$, $P =\iota= P'$):

Definition 1 *For injective mappings g, g', set A, and for P, P', and ι, and c :*

- $g' \leq g$ *wrt* A, *iff* $dom(g') = dom(g) \cup A$, *and* $\forall y \in dom(g) : g'(y) = g(y)$.
- $g' \preceq g$ *wrt* A, *iff* $dom(g') = dom(g) \cup A$, *and* $\forall y \in dom(g) \setminus A : g'(y) = g(y)$.
- $P =\iota= P'$ *iff* $\forall c : P(c) = P'(c)$, *and* $\forall \iota' \in dom(P) \setminus \{\iota\} : P(\iota') = P'(\iota')$.
- $P =c= P'$ *iff* $\forall c' \neq c : P(c') = P'(c')$, *and* $\forall \iota \in dom(P) : P(\iota) = P'(\iota)$.

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

$$W \vdash P' \leq P$$

c.f. figure 2, and is defined in the global context of a W which expresses the environment (possibly a file system) from which classes are loaded.

In more detail, $W \vdash P' \leq P$ if: 1) P' is in the reflexive, transitive closure of the relation. 2) P' and P are identical up to c, a raw class read from W whose superclass (c_s) is already in P. 3) P' and P are identical up to class c, and a) the field layout of c extends that of c_s and fields introduced by c get fresh offsets, b) the method layout of c extends that of c_s, c) all methods in c which override (have the same signature as) methods in c_s are mapped to new addresses. 4) P' and P are identical up to address ι, and $P(\iota')$ contains the verified/jit-compiled version of the method at $P(\iota)$.

The first rule of figure 3 says that programs may be extended at any time. The second rule allows linking exceptions to be thrown at any time. This is, of course, highly non-deterministic, and does not prohibit linking phases or errors even if unnecessary.

3.6 Evaluation is not directly affected by dynamic linking. It is described by the third through eighth rule in figure 3.

Creation of a new object, `new` c, allocates fresh addresses for the fields of c at the corresponding offsets, initializing them with **0**. The auxiliary function which collects

$$\frac{\begin{array}{l}P(c) = \langle \_, \tau^F, \_, \_\rangle \\ \tau^F(f,t) = \kappa\end{array}}{.f[c,t] \rightsquigarrow_P [\kappa]} \qquad \frac{\begin{array}{l}P(c) = \langle \_, \_, \tau^M, \_\rangle \\ \tau^M(m, t_r, t_p) = \kappa\end{array}}{.m[c, t_r, t_p] \rightsquigarrow_P [\kappa]} \qquad \frac{}{a \rightsquigarrow_P a}$$

Figure 4. Offset calculation.

the field offsets from all superclasses:

$$\mathit{FdOffs}(P,c) = \bigcup_{P \vdash c \leq c'} \mathit{rng}(P(c') \downarrow_2)$$

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

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 call and field access for **0** throw a `nllPExc`, as described in the sixth rule of the figure.

Execution is propagated to its context, as described in the seventh rule. Both link related, and 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:

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

3.7 Offset Calculation replaces a symbolic reference through an offset, and has format

$$a \rightsquigarrow_P a'$$

where a represents a field or method annotation. Figure 4 says that for fields, we look up the name of the field and its type in the class, whilst for methods we look up the name, argument type and result type in the class. The last rule allows a to be left unmodified.

The last rule in 3 allows offset calculation to happen during execution, as in Java. For this, we have defined appropriate notion of offset calculation contexts as

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

Offset calculation also happens during jit-compilation, (figure 5) thus modelling C#. Combining this with the rule that leaves offsets unmodified we model Java verification which does not calculate the offsets.

3.8 Verification and Jit-Compilation We describe the similarities between Java verification and C# jit-compilation through verification/jit-compilation, in fig. 5:

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

$$\frac{}{P, \mathtt{this} \leadsto_{W,E} P, \mathtt{this}, E(\mathtt{this})} \qquad \frac{P, c, c \leadsto_{W} P'}{P, \mathtt{new}\ c \leadsto_{W,E} P', \mathtt{new}\ c, c}$$

$$P, \mathtt{y} \leadsto_{W,E} P, \mathtt{y}, E(\mathtt{y}) \qquad P, 0 \leadsto_{W,E} P', 0, c$$

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

$$\frac{\begin{array}{l} P, e_1 \leadsto_{W,E} P_1, e_1', t_1 \\ P_1, e_2 \leadsto_{W,E} P_2, e_2', t_2 \\ P_2, t_1, c \leadsto_{W} P_3 \\ P_3, t_2, t_p \leadsto_{W} P' \\ .m[c, t_r, t_p] \leadsto_{P'} ma \end{array}}{P, e_1.m[c, t_r, t_p](e_2) \leadsto_{W,E} P', e_1'\, ma(e_2'), t_r} \qquad \frac{\begin{array}{l} W \vdash P'' \leq P \\ P'', e \leadsto_{W,E} P', e', t \end{array}}{P, e \leadsto_{W,E} P', e', t}$$

$$\frac{\begin{array}{l} W \vdash P' \leq P \\ P' \vdash t' \leq t \end{array}}{P, t', t \leadsto_{W} P'} \qquad \frac{}{P, t, t \leadsto_{W} P}$$

Figure 5. Verification and Jit-compilation.

which transforms expression e to e', type checks e to type t, and possibly extends the program P to P'. The process takes place in an environment E which maps `this` and the parameter `y` to types, *i.e.,* $E : \{\ \mathtt{this}, \mathtt{y}\ \} \rightarrow \mathit{Typ}$, and in the global context W.

The parameter `y` 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 call 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 call has type t_r, the result type of the annotation. The symbolic annotation may be replaced by an offset, thus modeling C# jit-compilation. Offset calculation also allows for the identity, thus modeling 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 sixth rule, which allows extension of the program at any time.

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 judgment:

$$P, t_1, t_2 \leadsto_{W} P'$$

also given in figure 5. Notice, that this judgment allows any identifier to be a subtype of itself even if not loaded - this follows the “optimistic” Java approach.

$$
\frac{
\begin{array}{l}
P(\texttt{Object}) = \langle \texttt{Object}, \_, \_, \_ \rangle \\
P \vdash c \le c' \quad \implies \quad c' \in dom(P) \ \text{ and } \ P \vdash c' \le c \implies c' = c \\
P(c) = \langle c', \tau^F, \tau^M, \tau^C \rangle \implies
\begin{cases}
c = c' \implies c = \texttt{Object} \\
\tau^F, \tau^M, \tau^C \text{ injective} \\
rng(\tau^M) = dom(\tau^C), \ \ rng(\tau^C) \subseteq dom(P \downarrow_2) \\
P \vdash c \le c_s, c \neq c_s \implies
\begin{cases}
P(c_s) = \langle \_, \tau^F_s, \tau^M_s, \_ \rangle \\
rng(\tau^F_s) \cap rng(\tau^F) = \emptyset \\
\tau^M \le \tau^M_s \text{ wrt some set}
\end{cases}
\end{cases} \\
\iota \in rng(P(c') \downarrow_4) \cap rng(P(c'') \downarrow_4) \implies
\begin{cases}
\exists c : \\
P \vdash c' \le c, \ \ P \vdash c'' \le c, \ \ \iota \in rng(P(c) \downarrow_4) \\
\iota \in rng(P(c''') \downarrow_4) \implies P \vdash c''' \le c \\
P(\iota) = e \implies
\begin{cases}
\exists e_0, \tau^M, \tau^C, m, t_r, t_p : \\
P, e_0 \rightsquigarrow_{\emptyset, \{\mathtt{this} \mapsto c, \mathtt{y} \mapsto t_p\}} P, e, t \\
P \vdash t \le t_r \\
P(c) = \langle \_, \_, \tau^M, \tau^C \rangle \\
\tau^C(\tau^M(m, t_r, t_p)) = \iota
\end{cases}
\end{cases}
\end{array}
}{\vdash P}
$$

Figure 6. Well-formed programs

$$
\frac{\begin{array}{l} P \vdash c' \le c \\ H(\iota) = c' \end{array}}{P, H \vdash \iota \lhd c}
\qquad
\frac{}{P, H \vdash \mathbf{0} \lhd c}
\qquad
\frac{\begin{array}{l} H(\iota) = c \\ P(c) = \langle \_, \tau^F, \_, \_ \rangle \\ \forall \kappa{:} TypeOfFd(P, c, \kappa) = t \implies P, H \vdash \iota + \kappa \lhd t \\ 1 \le \kappa \le max(FdOffs(P, c)) \implies H(\iota + \kappa) \notin ClassId \end{array}}{P, H \vdash \iota}
$$

$$
\frac{H(\iota) \in ClassId \implies P, H \vdash \iota}{P \vdash H}
$$

Figure 7. Conformance

4 Soundness and Equivalence of Strategies

The judgment $\vdash P$ defined in fig. 6 guarantees that program P is well formed, *i.e.,* that 1) the class `Object` is defined and has itself as a superclass, 2) all superclasses are present, and the subclass relationship is acyclic except for `Object`, 3) for any laid out class c with superclass c_s the fields and methods have distinct offsets, the methods defined in c_s have the same offsets in c, and 3) all the methods defined in c_s have the same offsets in c, and 4) all method bodies which are considered as already verified/jit-compiled, *i.e.,* for which P (ι)=e, can be verified/jit compiled, albeit without program extension, and therefore in the empty global context, $\emptyset$.

Figure 7 defines conformance. The judgment $P, H \vdash \iota$ expresses that the object stored at ι conforms to its class, c, as stored in H (ι). 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 judgment $P \vdash H$ requires all objects to conform to their class, and (implicitly) that the class of any objects stored in H is defined in P.

$$\frac{}{\begin{array}{l}P,H \vdash \mathbf{0} : c\\ P,H \vdash \texttt{new}\ c : c\end{array}} \qquad \frac{\begin{array}{l}P,H \vdash \iota\\ P \vdash c \leq c'\\ H(\iota) = c\end{array}}{P,H \vdash \iota : c'} \qquad \frac{\begin{array}{l}P,H \vdash e : c'\\ P \vdash c' \leq c\end{array}}{P,H \vdash e.f[c,t] : t} \qquad \frac{\begin{array}{l}P,H \vdash e : c\\ \mathit{TypeOfFd}(P,c,\kappa) = t\end{array}}{P,H \vdash e[\kappa] : t}$$

$$\frac{\begin{array}{l}P,H \vdash e\,fa : t\\ P,H \vdash e' : t'\\ P \vdash t' \leq t\end{array}}{P,H \vdash e\,fa = e' : t} \qquad \frac{\begin{array}{l}P,H \vdash e_1 : c_1\\ P,H \vdash e_2 : t_2\\ P \vdash c_1 \leq c\\ P \vdash t_2 \leq t_p\end{array}}{P,H \vdash e_1.m[c,t_r,t_p](e_2) : t_r} \qquad \frac{\begin{array}{l}P,H \vdash e_1 : c_1\\ P,H \vdash e_2 : t_2\\ P \vdash t_2 \leq t_p\\ P(c_1) = \langle \_,\_,\tau^M,\_\rangle\\ \tau^M(\langle ..., t_r, t_p\rangle) = \kappa\end{array}}{P,H \vdash e_1[\kappa](e_2) : t_r}$$

Figure 8. Types of runtime expressions.

Notice, that **0** conforms to any class, allowing fields initialized to **0**, to belong even to a class that has not been loaded yet.

Types for runtime expressions are given by judgment $P, H \vdash e : t$, from fig. 8, 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 `y`), and the program is *not* extended.

Runtime expressions containing field access offsets are typed using:

$$\mathit{TypeOfFd}(P,c,\kappa) \;=\; t \text{ if } P(c')\downarrow_2 (\_,t) = \kappa \;\text{ for } P \vdash c \leq c', \quad \epsilon \text{ otherwise}$$

The above, and the inverse layout function for runtime types of method calls, are well-defined in well formed programs, because layout functions are injective.

In the longer version we prove that verification/jit-compilation and execution extend programs. Subtyping, conformance of heap, runtime types, verification of expressions, or well-formedness of program, established in a program P are preserved in an extending program P' . Therefore, execution of *any* expression preserves well-formedness of programs. Finally, a verified expression preserves its runtime type, when the receiver and argument have been replaced by addresses of appropriate class.

In theorem 1 we prove subject reduction which guarantees that the heap H' preserves conformance, uninitialized parts of the store are never dereferenced, and the expression preserves its type. In theorem 2 we prove that nondeterminism does *not* affect the result of evaluations which do not throw link related exceptions, provided we operate in the same global context W.

Theorem 1 *If* $P \vdash H$, *and* $\vdash P$, *and* $\;P, H \vdash e : t$, *and* $\;P, H, e \leadsto_W P', H', e'$ *then*

$P' \vdash H'$, *and*

if e' *does not contain an exception, then* $\quad \exists\, t' : \; P', H' \vdash e' : t', \; P' \vdash t' \leq t$.

Theorem 2 *For* e, P, P', P'', H, H', H'', ι , *and* $\nu, \nu' \in \mathbb{N} \cup \{\texttt{nllPExc}\}$, *if*:

$$P, H, e \leadsto^*_W P', H', \nu, \quad \text{and} \quad P, H, e \leadsto^*_W P'', H'', \nu',$$

then:

$$\nu = \nu', H' = H'' \quad \textit{up to renaming of addresses.}$$

Theorem 2 does *not* apply for intermediate results, nor if ν were a link related exception – counterexamples apeared in section 2.

5 Conclusions, related work, and further work

Dynamic linking is a relatively new, very powerful language feature with complex semantics, which needs to be well understood. Our model is simple, especially considering the complexity of the feature, and compared to an earlier model for Java [4].

We have achieved simplicity through many iterations, and through the choice of appropriate abstractions: 1) we do not distinguish the causes of link related exceptions, 2) we allow link-related exceptions to be thrown at *any* time of execution, even when there exist other, legal evaluations, 3) we do not prescribe at which point of execution the program will be extended, and so allow "unnecessary" loading, verification or jit-compilations, 4) we combine in the concept of "program" loaded, verified, and laid out code, 5) we represent programs through mapping rather than texts or data structures. Most of these abstractions were introduced primarily to allow the model to serve for both Java and for C#, and had the agreeable effect of significant simplification.

Non-determinism seems to have been in the the Java designers' minds: the specification [17], sect. 12.1.1 requires resolution errors to be thrown only when linking actions related to the error are required. Through non-determinism we distilled the main ingredients of dynamic linking in both languages, and their dependencies. 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 [24,11]. Dynamic loading in Java is formalized in [14], while problems with security in the presence of multiple loaders are reported in [23], a solution presented in [16], which is found flawed and improved upon in [21]. Type safety for a substantial subset of the .NET IL is proven in [12].

The semantics of linking is studied in [2]. Module interconnection languages, and mixins [1,8,6] give explicit control of program composition at source code level.

Dynamic linking gave rise to the concept of binary compatible changes, [9], and [17], sect. 13, *i.e.,* changes which do not introduce more linking errors than the original code; the concept is explored in [5]. Tools that load most recent binary compatible versions of code were developed for Java [22] and C# [15]. Current JVMs go even further, and support replacing a class by a class of the same signature, as a "fix-and-continue" feature [3]. Dynamic software updates [13] support type safe dynamic reloading of code whose type may have changed, while the system is running.

Further work includes a better understanding of binary compatible library developments, extension of the model to allow verification also posting constraints, as suggested in [21], or to allow field lookup based on superclass's tables as in some of JVMs, 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 We are indebted to Vladimir Jurisic, Davide Ancona, Elena Zucca, Christopher Anderson, and Mark Skipper for suggestions and feedback.

References

1. Davide Ancona and Elena Zucca. A calculus of module systems. *Journal Functional Programming*, 12(3):91–132, 2002.
2. Luca Cardelli. Program Fragments, Linking, and Modularization. In *POPL'97 Proceedings*, January 1997.
3. Mikhail Dimitriev. Hotspot Technology Application for Advanced Profiling. In *ECOOP USE Workhop*, June 2002.
4. Sophia Drossopoulou. An Abstract model of Java dynamic Linking and Loading. In Robert Harper, editor, *Types in Compilation, Third International Workshop, Revised Selected Papers*. Springer, 2001.
5. Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A Fragment Calculus - towards a model of Separate Compilation, Linking and Binary Compatibility. In *LICS Proceedings*, 1999.
6. Dominic Duggan. Sharing in Typed Module Assembly Language. In *Preliminary Proceedings of the Third Workshop on Types in Compilation (TIC 2000)*. Carnegie Mellon, CMU-CS-00-161, 2000.
7. G. Fenton and E. Felton. *Securing Java Getting Down to Business with Mobile Code*. John Wiley and Sons, 1999.
8. Kathleen Fisher, John Reppy, and Jon Riecke. A Calculus for Compiling and Linking Classes. In *ESOP Proceedings*, March 2000.
9. Ira Forman, Michael Conner, Scott Danforth, and Larry Raper. Release-to-Release Binary Compatibility in SOM. In *OOPSLA Proceedings*, October 1995.
10. Stephen N. Freund and J. C. Mitchell. A Formal Framework for the Java Bytecode Language and Verifier. In *OOPSLA Proceeedings*, November 1999.
11. Stephen N. Freund and J. C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. In *OOPSLA Proceeedings*, October 1998.
12. Andrew Gordon and Don Syme. Typing a multi-language intermediate code. In *Principles of programming Languages 2001*, pages 248–260. ACM Press, 2001.
13. Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic Software Updating. In *Programming Language Design and Implementation*. ACM, 2001.
14. Thomas Jensen, Daniel Le Metyayer, and Tommy Thorn. A Formalization of Visibility and Dynamic Loading in Java. In *IEEE ICCL*, 1998.
15. V. Jurisic. Deja-vu.NET: A Framework for Evolution of Component Based Systems. http://www.doc.ic.ac.uk/˜ajf/Teaching/Projects/DistProjects.html, June 2002.
16. Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java^{TM} Virtual Machine. In *OOPSLA Proceedings*, October 1998.
17. Tim Lindholm and Frank Yellin. *The Java Virtual Machine*. Addison-Wesley, 1999.
18. Type Safety and Security. In *MSDN Magazine*, 2001.
19. M. Pietrek. Avoiding DLL Hell: Introducing Application Metadata in the Microsoft .NET Framework. In *MSDN Magazine*, msdn.microsoft.com/, October 2000.
20. S. Pratschner. Simplifying Deployment and Solving DLL Hell with the .NET Framework. msdn.microsoft.com/, November 2001.
21. Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A Formal Specification of Java^{TM} Class Loading. In *OOPSLA'2000*, November 2000.
22. C. Sadler S. Eisenbach and S. Shaikh. Evolution of Distributed Java Programs. In *IEEE Working Conf on Component Deployment*, June 2002.
23. Vijay Saraswat. Java is not type-safe. Technical report, AT&T Rresearch, 1997. http://www.research. att.comp/˜vj/bug.html.
24. Raymie Stata and Martin Abadi. A Type System For Java Bytecode Subroutines. In *POPL'98 Proceedings*, January 1998.

Appendix: Overview of terms and judgments of this paper

e	expressions	fig. 1
t	types	fig. 1
ι	addresses	fig. 1
κ	offsets	fig. 1
`nllPExc`	the null-pointer exception	fig. 1
`lnkExc`	link-related exception, *e.g.,* verification, load err. *etc*	fig. 1
fa, ma, a	field, method, or any annotation	fig. 1
δ^F	field descriptions	fig. 1
δ^M	method descriptions	fig. 1
τ^F	field layout tables	fig. 1
τ^M	method layout tables	fig. 1
τ^C	code tables	fig. 1
H	heaps	sec. 3
E	environment giving types to receiver and argument	sec. 3
$\sqsubset\cdot\sqsupset^{exe}$	execution context	sect. 3
$\sqsubset\cdot\sqsupset^{off}$	offset calculation context	sect. 3
$P, H, e \leadsto_W P', H', e'$	execution in global context W	fig 3
$a \leadsto_P a'$	offset calculation	fig. 4
$P, e \leadsto_{W,E} P', e', t$	verification or jit-compilation	fig. 5
$P, t', t \leadsto_W P'$	t' is a subtype of t, while extending program P to P'	fig. 5
$W \vdash P' \leq P$	program P' extends program P in global context W	fig. 2
$P \vdash t' \leq t$	in program P the type t' is a subtype of t	fig. 1
$\vdash P$	well formed program	fig. 6
$P \vdash H$	well formed heap H for the program P	fig. 7
$P, H \vdash e : t$	runtime expression e has type t in the context of P and H	fig. 8
$P, H \vdash \iota \triangleleft c$	ι conforms class c, or subclass	fig. 7
$g' \leq g$ wrt A	mapping g' injectively extends g into set A, preserving $dom(g)$	def. 1
$g' \preceq g$ wrt A	mapping g' injectively extends g into set A, preserving $dom(g) \setminus A$	def. 1
$P =\iota= P'$	P and P' agree up to address ι	def. 1
$P =c= P'$	P and P' agree up to class c	def. 1
$FdOffs(P, c)$	the set of all offsets allocated for the fields of c in P	page 10
$TypeOfFd(P, c, \kappa)$	the type of the field contained at the offset κ of c in P	page 13