Flexible, source level dynamic linking and re-linking *
– Work in Progress –

Sophia Drossopoulou, Susan Eisenbach,
Imperial College London

Abstract

We give a formal semantics for dynamic linking and re-linking of code. The semantics is at source language level, and allows linking at a finer grain than current Java or C# implementations: Besides supporting the loading and verification of classes interleaved with program execution, it also allows type-safe removal and replacement of classes, fields and methods. Such extended features support unanticipated software evolution.

1 Introduction

Dynamic linking was introduced into programming languages to support implicit linking of the newest version of libraries. Partially linked code links further code on the fly, as needed, and thus, most recent updates are automatically available to end-users, without the requirement of further compilations or re-linking.

The remit has recently been extended, to support requirements from unanticipated software evolution, whereby code may need to be *updated during* program execution. To address such requirements, new features of the JVM allow the replacement of a class by a class of the same signature, as a "fix-and-continue" feature [5], while in [11] dynamic software updates support type safe dynamic reloading of code whose type may have changed, while the system is running. Also, [2] suggest dynamic linking of modules, with the support of multiple versions of the same module. Finally, [1] suggest a calculus for dynamic linking independently of the programming language and environment.

We present the semantics for a high level language similar to Java or C# with extended dynamic linking. In contrast to earlier formalizations of dynamic linking [12, 8], our model is purely at source language level. Namely, although dynamic linking is usually described at a "low level" [10], in terms of generated *.class files, its effects are visible to the source language programmer, as shown *e.g.*, through a sequence of examples in [6, 9]. This is why, in our view, an explanation at source language level is very important. In order to simplify the exposition, we do not deal with overloading or field shadowing.

In more detail, the model we suggest allows dynamic linking at a very fine grain:

DL1 Method signatures may be loaded before verification of their corresponding bodies,

DL2 Method bodies need be verified only before execution[1],

DL3 Classes may be loaded before their fields/methods,

DL4 Classes, methods and fields may be loaded "piecemeal",

DL5 Classes, methods, and fields may be removed, if not needed,

DL6 Method bodies may be replaced.

*This work partly supported by DART, EU project IST-2001-33477.

[1]This is similar to C# dynamic linking.

$$\frac{\chi \vdash \pi' \leq \pi \quad \mathtt{e}, \pi', \sigma, \chi \rightsquigarrow r, \pi'', \chi'}{\mathtt{e}, \pi, \sigma, \chi \rightsquigarrow r, \pi'', \chi'} \; lnk$$

$$\frac{}{\mathtt{e}, \pi, \sigma, \chi \rightsquigarrow \mathtt{LnkExc}, \pi, \chi} \; exc$$

$$\frac{\chi \ll \chi'' \; @ \; \iota \quad \iota \text{ not reachable from } \sigma \text{ in } \chi \quad \mathtt{e}, \pi, \sigma, \chi'' \rightsquigarrow r, \pi', \chi'}{\mathtt{e}, \pi, \sigma, \chi \rightsquigarrow r, \pi', \chi'} \; gbgCllt$$

$$\frac{}{\mathtt{y}, \pi, \sigma, \chi \rightsquigarrow \sigma(\mathtt{y}), \pi, \chi \quad \mathbf{this}, \pi, \sigma, \chi \rightsquigarrow \sigma(\mathbf{this}), \pi, \chi \quad r, \pi, \sigma, \chi \rightsquigarrow r, \pi, \chi} \; var$$

$$\frac{\mathtt{e}, \pi, \sigma, \chi \rightsquigarrow \iota, \pi', \chi' \quad \mathtt{e}', \pi', \sigma, \chi' \rightsquigarrow \kappa, \pi'', \chi'' \quad \chi''(\iota)(\mathtt{f}) = \kappa', \quad \chi''' = \chi''[\iota \mapsto \chi''(\iota)[\mathtt{f} \mapsto \kappa]]}{\mathtt{e.f} := \mathtt{e}', \pi, \sigma, \chi \rightsquigarrow \kappa, \pi'', \chi'''} \; fldAss$$

$$\frac{\mathtt{e}, \pi, \sigma, \chi \rightsquigarrow \iota, \pi', \chi'}{\mathtt{e.f}, \pi, \sigma, \chi \rightsquigarrow \chi'(\iota)(\mathtt{f}), \pi', \chi'} \; fld$$

$$\frac{\mathtt{e}, \pi, \sigma, \chi \rightsquigarrow \iota, \pi_1, \chi_1 \quad \mathtt{e}_i, \pi_i, \sigma, \chi_i \rightsquigarrow \kappa_i, \pi_{i+1}, \chi_{i+1} \quad 1 \leq i \leq n \quad \chi_{n+1}(\iota) = (\mathtt{c}, \ldots) \quad \mathcal{MB}(\pi_{n+1}, \mathtt{c}, \mathtt{m}) = \mathtt{e}' \quad \phi = \mathbf{this} \mapsto \iota, \mathtt{y_1} \mapsto \kappa_1, \ldots \mathtt{y_n} \mapsto \kappa_n \quad \mathtt{e}', \pi_{n+1}, \sigma \cdot \phi, \chi_{n+1} \rightsquigarrow \kappa, \pi', \chi'}{\mathtt{e}.m(\mathtt{e}_1, \ldots \mathtt{e}_n), \pi, \sigma, \chi \rightsquigarrow \kappa, \pi', \chi'} \; call$$

$$\frac{\mathcal{FS}(\pi, \mathtt{c}) = \{ \mathtt{f}_1, \ldots \mathtt{f}_r \} \quad \iota \text{ is fresh in } \chi \quad \chi' = \chi[\iota \mapsto (\mathtt{c}, \mathtt{f}_1 \mapsto \mathbf{0} \ldots \mathtt{f_r} \mapsto \mathbf{0})]}{\mathbf{new}\ \mathtt{c}, \pi, \sigma, \chi \rightsquigarrow \iota, \pi, \chi'} \; new$$

Figure 1: Execution.

DL1-DL4 represent a refinement of dynamic linking as currently implemented. **DL5-DL6** are even more "dynamic", and come at a higher implementation cost, as they require re-checking the entire program. **DL5-DL6** have been suggested in order to support on-the-fly code replacement, and thus, unanticipated software evolution.

Furthermore, the fact that we model dynamic linking at source language level, *imposes* the requirement for **DL1**and **DL2**: Namely, when verifying a method body, `e`, we need to find the signatures of any methods called within `e`; for this, we load the method signatures for all called methods. But, we do *not* verify their corresponding method bodies, because if we did, we would obtain total, eager, verification *before* execution.[2]

Notation All mappings in our model are partial, and ϵ represents undefined. A function f' *extends* another function f at a, formally $f' \ll f \; @ \; a$, if f' is defined on one more element than f, *i.e.,* iff $f' \downarrow_{dom(f)} = f$ and $dom(f') = dom(f) \uplus \{ a \}$.

2 Syntax, runtime model, and execution

Expressions in our minimal language are method call, field assignment, field access, object creation, the receiver **this**, a parameter identifier $\mathtt{y}_i$ ($i \in \mathbb{N}$) or null indicated by $\mathbf{0}$. Class names are indicated by `c`, `c`$'$ *etc*, method names are `m`, `m`$'$, *etc*, and field identifiers are `f`, `f`$'$.

$\mathtt{e} \in \mathtt{Exp} \quad ::= \quad \mathtt{e.m(e^*)} \mid \mathtt{e.f} := \mathtt{e} \mid \mathtt{e.f} \mid \mathbf{new}\ \mathtt{c} \mid \mathbf{this} \mid \mathtt{y_i} \mid \mathbf{0}$

[2]This problem does not appear in Java and C# dynamic linking, because bytecode is "enriched" with the method signature for called methods; therefore, verification takes place under the assumption that the classes contain the corresponding methods, and if they do not, then an offset calculation error is thrown. In that sense, verification in our model is more eager than in Java and C#.

We represent dynamic linking through "extensions" to the underlying program π [3]. Therefore, the operational semantics rewrites tuples of expressions, programs, stacks and heaps into tuples of results, programs and heaps.

$$
\begin{array}{rcl}
\rightsquigarrow & : & \texttt{Exp} \times \texttt{Prg} \times \texttt{Stack} \times \texttt{Heap} \longrightarrow \mathcal{R} \times \texttt{Prg} \times \texttt{Heap} \\
\sigma \in \texttt{Stack} & = & (\,\texttt{StackFrm}\,)* \\
\phi \in \texttt{StackFrm} & = & (\,\{\,\mathbf{this}\,\} \uplus \texttt{Param}\,) \longrightarrow \mathbb{N} \\
\chi \in \texttt{Heap} & = & \mathbb{N}^{+} \longrightarrow \texttt{Obj} \\
o \in \texttt{Obj} & = & \texttt{ClssId} \times \texttt{FieldMap} \\
fm \in \texttt{FieldMap} & = & \texttt{FldId} \longrightarrow \mathbb{N} \\
\iota \in \mathbb{N}^{+} & & \\
\kappa \in \mathbb{N} & & \\
r \in \mathcal{R} & = & \mathbb{N} \cup \{\,\texttt{nllPtrExc}, \texttt{StuckExc}, \texttt{LnkExc}\,\}
\end{array}
$$

Stacks, σ, are sequences of stack frames, ϕ, which map **this** to an address, and the parameters $\texttt{y}_1$, ... $\texttt{y}_n$ to addresses or null. We use the shorthand $\sigma\,(\texttt{z})$, for $\phi_n\,(\texttt{z})$, when ϕ_n is the top of σ (*i.e.*, $\sigma = \phi_1 ... \phi_n$). Heaps, χ, map addresses to objects. The notation $o = (\,\texttt{c},\ fm\,)$, stands for an object of class c, with fields described by the mapping fm. For such an object, the field lookup $o(\texttt{f})$ is a shorthand for $o{\downarrow_2}(\texttt{f})$, and field update $o[\texttt{f} \mapsto \kappa]$ is a shorthand for $(\,o\downarrow_1, o\downarrow_2[\texttt{f} \mapsto \kappa]\,)$. Results, r, are either addresses, or null ($\mathbf{0}$), or exceptions. Addresses, ι , are positive numbers.

Figure 1 contains the operational semantics of the language. Rule *lnk* replaces a prograqm by an extending program *at any time* during execution, while *exc* allows link related exceptions to be thrown *at any time* during execution. We thus have a highly non-deterministic model, similarly to [8]. Rule *gbgCllt* allows for garbage collection – the appendix contains the definition for reachable addresses. Garbage collection is important in our study, as classes may be removed when there are no objects of that class. Garbage collection is defined in a style similar to that from [3].

The remaining rules are standard for the operational semantics of such a small object oriented language, and are similar to previous work [7]: Rule *var* describes the lookup of parameters, the receiver, or a value. Rules *fld* and *fldAss* describe field lookup and field update. Rule *call* describes method call: a new frame ϕ is pushed onto the stack, which is discarded after execution of the method body $\texttt{e}'$. Rule *new* describes object creation, where all fields of its class are initialized with $\mathbf{0}$. In the appendix we give rules for null pointer exceptions and for stuck execution.

The different kinds of exception and the guarantees of Soundness Our calculus allows three kinds of exception: *Null pointer* exceptions, `nllPtrExc`, are thrown when a null-pointer is de-referenced to access a field or a method; they are propagated to the context as well. *Stuck* exceptions, `stuckExc`, are thrown when an object is de-referenced ro access a non-existing field, or when a non-existing method is called; they are also propagated to the context. *Link* exceptions, `LnkExc`, may be thrown at any point during execution, and thus they need not be propagated to the context. They represent link-related exceptions, *i.e.*, verification errors, class-not-found errors, class-circularity-errors, load errors *etc*.[4]

Soundness, as stated in section 4, guarantees that execution will not get stuck, *i.e.*, that that execution will not attempt to access a non-existing field or method. The case of an "absent method" is slightly subtle: If the dynamic class of the receiver of a method call of `m` contains a method signature for that `m`, but contains no method body, then that represents the case where the method body could not be verified, and therefore it is a link related error. If, on the other hand, the dynamic class of the receiver contains no method signature for `m`, then, this corresponds to the case where the class has no such method, and therefore it is a `stuckExc`. More details are in the appendix.

[3] Note, that "program extensions" stands for all the dynamic linking steps from **DL1**- **DL6**, *i.e.*, even such that remove classes or methods

[4] Of course, we do not model the `FldAbsent` and `MethodAbsent` errors of Java and C#, since these are related to the "enriched" bytecode, and offset calculations.

refl

$$\frac{}{\chi \vdash \pi \leq \pi}$$

trans

$$\frac{\chi \vdash \pi' \leq \pi'' \quad \chi \vdash \pi'' \leq \pi}{\chi \vdash \pi' \leq \pi}$$

ldClss

$$\frac{\pi'_s \ll \pi_s \ @ \ \mathtt{c} \quad \pi_s(\pi'_s(\mathtt{c})) \neq \epsilon}{\chi \vdash (\pi'_s, \pi_f, \pi_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$$

ldFld

$$\frac{\pi'_f(\mathtt{c}) \ll \pi_f(\mathtt{c}) \ @ \ \mathtt{f} \quad \pi_s(\mathtt{c}) \neq \epsilon \quad \chi(\iota) = (\ \mathtt{c}', \_\ ) \implies \pi \not\vdash \mathtt{c}' \leq \mathtt{c} \quad \pi \vdash \mathtt{c} \not\approx \mathtt{c}' \implies \pi'_f(\mathtt{c})(\mathtt{f}) = \epsilon}{\chi \vdash (\pi_s, \pi'_f, \pi_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$$

ldMthSig

$$\frac{\pi'_{ms} \ll \pi_{ms} \ @ \ (\mathtt{c}, \mathtt{m}) \quad \pi_s(\pi'_s(\mathtt{c})) \neq \epsilon \quad \pi \vdash \mathtt{c}' \leq \mathtt{c} \implies \pi \vdash \pi_{ms}(\mathtt{c}', \mathtt{m}) \leq \pi'_{ms}(\mathtt{c}, \mathtt{m}) \quad \pi \vdash \mathtt{c} \leq \mathtt{c}' \implies \pi \vdash \pi'_{ms}(\mathtt{c}, \mathtt{m}) \leq \pi_{ms}(\mathtt{c}', \mathtt{m})}{\chi \vdash (\pi_s, \pi_f, \pi'_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$$

verMthBdy

$$\frac{\pi'_{mb} \ll \pi_{mb} \ @ \ (\mathtt{c}, \mathtt{m}) \quad \pi_{ms}(\mathtt{c}, \mathtt{m}) = \mathtt{c}_1, \ldots \mathtt{c}_{n+1}, \quad n \geq 0 \quad \pi'_{mb}(\mathtt{c}, \mathtt{m}) = \mathtt{e} \quad \pi, (\mathbf{this} \mapsto \mathtt{c}, \mathtt{y_1} \mapsto \mathtt{c_1}, \ldots \mathtt{y_n} \mapsto \mathtt{c_n}) \vdash \mathtt{e} : \mathtt{c}' \quad \pi \vdash \mathtt{c}' \leq \mathtt{c}_{n+1}}{\chi \vdash (\pi_s, \pi_f, \pi_{ms}, \pi'_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$$

rmFld

$$\frac{\pi_f(\mathtt{c}) \ll \pi'_f(\mathtt{c}) \ @ \ \mathtt{f} \quad \pi_{ms}(\mathtt{c}', \mathtt{m}) = \mathtt{c}_1, \ldots \mathtt{c}_{n+1}, \ \ \pi_{mb}(\mathtt{c}', \mathtt{m}) = \mathtt{e} \implies \pi', (\mathbf{this} \mapsto \mathtt{c}', \mathtt{y_1} \mapsto \mathtt{c_1}, \ldots \mathtt{y_n} \mapsto \mathtt{c_n}) \vdash \mathtt{e} : \mathtt{c}'' \quad \pi \vdash \mathtt{c}'' \leq \mathtt{c}_{n+1}}{\chi \vdash (\pi_s, \pi'_f, \pi_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$$

rmMthSig

$$\frac{\pi_{ms} \ll \pi'_{ms} \ @ \ (\mathtt{c}, \mathtt{m}) \quad \pi_{mb}(\mathtt{c}, \mathtt{m}) = \epsilon \quad \pi_{ms}(\mathtt{c}', \mathtt{m}') = \mathtt{c}_1, \ldots \mathtt{c}_{n+1}, \ \ \pi_{mb}(\mathtt{c}', \mathtt{m}') = \mathtt{e} \implies \pi', (\mathbf{this} \mapsto \mathtt{c}', \mathtt{y_1} \mapsto \mathtt{c_1}, \ldots \mathtt{y_n} \mapsto \mathtt{c_n}) \vdash \mathtt{e} : \mathtt{c}'' \quad \pi \vdash \mathtt{c}'' \leq \mathtt{c}_{n+1}}{\chi \vdash (\pi_s, \pi_f, \pi'_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$$

rmMthBdy

$$\frac{\pi_{mb} \ll \pi'_{mb} \ @ \ (\mathtt{c}, \mathtt{m})}{\chi \vdash (\pi_s, \pi_f, \pi_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi'_{mb})}$$

rplMthBdy

$$\frac{\pi'_{mb} = \pi_{mb}[(\mathtt{c}, \mathtt{m}) \mapsto \mathtt{e}] \quad \pi_{ms}(\mathtt{c}, \mathtt{m}) = \mathtt{c}_1, \ldots \mathtt{c}_{n+1}, \quad n \geq 0 \quad \pi, (\mathbf{this} \mapsto \mathtt{c}, \mathtt{y_1} \mapsto \mathtt{c_1}, \ldots \mathtt{y_n} \mapsto \mathtt{c_n}) \vdash \mathtt{e} : \mathtt{c}' \quad \pi \vdash \mathtt{c}' \leq \mathtt{c}_{n+1}}{\chi \vdash (\pi_s, \pi_f, \pi_{ms}, \pi'_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$$

rmClss

$$\frac{\pi_s \ll \pi'_s \ @ \ \mathtt{c} \quad \chi(\iota) = (\ \mathtt{c}', \_\ ) \implies \pi \not\vdash \mathtt{c} \leq \mathtt{c}' \quad \pi_s(\mathtt{c}') \neq \mathtt{c} \ \ \text{for all } \mathtt{c}' \quad \pi_{ms}(\mathtt{c}, \mathtt{m}) = \pi_f(\mathtt{c}, \mathtt{f}) = \epsilon \ \ \text{for all } \mathtt{m}, \mathtt{f} \quad \pi_{ms}(\mathtt{c}', \mathtt{m}) = \mathtt{c}_1, \ldots \mathtt{c}_{n+1}, \ \ \pi_{mb}(\mathtt{c}, \mathtt{m}) = \mathtt{e} \implies \pi', (\mathbf{this} \mapsto \mathtt{c}', \mathtt{y_1} \mapsto \mathtt{c_1}, \ldots \mathtt{y_n} \mapsto \mathtt{c_n}) \vdash \mathtt{e} : \mathtt{c}'' \quad \pi \vdash \mathtt{c}'' \leq \mathtt{c}_{n+1}}{\chi \vdash (\pi'_s, \pi_f, \pi_{ms}, \pi_{mb}) \leq (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})}$$

Figure 2: Program extension

3 Programs and program extension

A program maps a class to its superclass, a class and a field identifier to its type, and class and method identifier to its signature, and method body:

$$
\begin{array}{lll}
\pi \in \texttt{Prg} = & \texttt{ClssId} \longrightarrow \texttt{ClssId} & \text{maps class to its superclass} \\
\times & \texttt{ClssId} \longrightarrow \texttt{FldId} \longrightarrow \texttt{ClssId} & \text{maps class and field name to type} \\
\times & \texttt{ClssId} \times \texttt{MthId} \longrightarrow (\texttt{ClssId}^{+}) & \text{maps class and meth. name to type} \\
\times & \texttt{ClssId} \times \texttt{MthId} \longrightarrow \texttt{Exp} & \text{maps class and meth. name to body}
\end{array}
$$

A program is therefore a tuple; we use the notation $(\pi_s, \pi_f, \pi_{ms}, \pi_{mb})$ to indicate the four components, and we use the naming convention that π_s, π_f, π_{ms}, and π_{mb} stand for mappings of the appropriate signatures.

We now define field and method look up functions:

$$
\begin{array}{lcl}
\mathcal{F} & : & \texttt{Prg} \times \texttt{ClssId} \times \texttt{FldId} \longrightarrow \texttt{ClssId} \\
\mathcal{F}s & : & \texttt{Prg} \times \texttt{ClssId} \times \texttt{FldId} \longrightarrow \mathcal{P}(\texttt{FldId}) \\
\mathcal{MS} & : & \texttt{Prg} \times \texttt{ClssId} \times \texttt{MthId} \longrightarrow \texttt{ClssId}^{*} \\
\mathcal{MB} & : & \texttt{Prg} \times \texttt{ClssId} \times \texttt{MthId} \longrightarrow \texttt{Exp}
\end{array}
$$

where, for $\pi = (\pi_s, \pi_f, \pi_{ms}, \pi_{mb})$, these functions are defined as follows:

$$
\mathcal{F}(\pi, \texttt{c}, \texttt{f}) = \begin{cases} \pi_f(\texttt{c})(\texttt{f}) & \text{if } \pi_f(\texttt{c})(\texttt{f}) \neq \epsilon \\ \mathcal{F}(\pi, \texttt{c}', \texttt{f}) & \text{if } \pi_f(\texttt{c})(\texttt{f}) = \epsilon, \text{ and } \texttt{c} \neq \pi_s(\texttt{c}) = \texttt{c}' \\ \epsilon & \text{otherwise.} \end{cases}
$$

$$
\mathcal{F}s(\pi, \texttt{c}) = \begin{cases} \{\ \texttt{f} \mid \mathcal{F}(\pi, \texttt{c}, \texttt{f}) \neq \epsilon\ \} & \text{if } \pi_s(\texttt{c}) \neq \epsilon \\ \epsilon & \text{otherwise} \end{cases}
$$

$$
\mathcal{MS}(\pi, \texttt{c}, \texttt{m}) = \begin{cases} \pi_{ms}(\texttt{c}, \texttt{m}) & \text{if } \pi_{ms}(\texttt{c}, \texttt{m}) \neq \epsilon \\ \mathcal{MS}(\pi, \texttt{c}', \texttt{m}) & \text{if } \pi_{ms}(\texttt{c}, \texttt{m}) = \epsilon, \text{ and } \texttt{c} \neq \pi_s(\texttt{c}) = \texttt{c}' \\ \epsilon & \text{otherwise.} \end{cases}
$$

$$
\mathcal{MB}(\pi, \texttt{c}, \texttt{m}) = \begin{cases} \pi_{mb}(\texttt{c}, \texttt{m}) & \text{if } \pi_{mb}(\texttt{c}, \texttt{m}) \neq \epsilon \\ \mathcal{MB}(\pi, \texttt{c}', \texttt{m}) & \text{if } \pi_{mb}(\texttt{c}, \texttt{m}) = \epsilon, \text{ and } \texttt{c} \neq \pi_s(\texttt{c}) = \texttt{c}' \\ \epsilon & \text{otherwise.} \end{cases}
$$

The judgments $\pi_s \vdash \texttt{c}' \leq \texttt{c}$ and $\pi \vdash \texttt{c}' \leq \texttt{c}$ mean that $\texttt{c}'$ is a subclass of $\texttt{c}$, while the judgments $\pi_s \vdash \texttt{c}' \not\leq \texttt{c}$ and $\pi_s \vdash \texttt{c}' \not\leq \texttt{c}$ mean that the two classes are distinct with one a subclass of the other. The four judgments are defined in the appendix.

The rules in figure 2 define when a program extends another program:

refl and *trans* say that the extension relationship is reflexive and transitive.

ldClss describes loading of a class `c`, provided that the superclass of the loaded class has a superclass defined in the old program, *i.e.*, for $\pi'_s(\texttt{c}) = \texttt{c}'$, we need $\pi_s(\texttt{c}') \neq \epsilon$. Note, that the class is loaded a an "empty shell" and its fields and methods may be loaded later, “piecemeal”, through rules *ldFld* and *ldMthSig*.

ldFld describes loading the type of a field `f` for a class `c`. It requires that no objects of that class or its subclasses exist on the heap[5], and that the field `f` has not been defined in any of the subclasses or superclasses of `c` [6].

[5] If we dropped this requirement, then at a later stage we could verify a method body that used the new field, although we might have objects in the heap, which did not contain that field

[6] In dynamic linking as in current JVMs and C# implementations, the situation is simpler. The requirement that there are no objects of subclass of `c` is implicitly guaranteed because classes are fully loaded before the first object is created, and the requirement that there is no other field `f` is any of the sub- or superclasses of `c` need not be imposed because field resolution takes the class defining the field into account.

ldMthSig describes loading the signature of a method `m` in a class `c`. It requires any signature for `m` in a subclass of `c` to be a "subsignature" of the newly loaded signature, and any signature for `m` in a superclass of `c` to be a "supersignature" of the newly loaded signature,[7] The "subsignature judgment $\pi \vdash \mathtt{c}'_1 ... \mathtt{c}'_n \leq \mathtt{c}_1 ... \mathtt{c}_n$ requires the argument types of the first to be supertypes of the corresponding argument types of the second, and the return type of the first to be a subtype of the return type of the second - or the second signature to be empty; it is defined in the appendix. Notice that it does *not* require the classes appearing in the signatures of methods or the types of fields to belong to classes that have already been loaded.

verMthBdy describes the verification of a method body. It requires the signature of the method to have already been loaded, and the body of the method to verify, *i.e.,* to be type correct, assuming that the parameters $\mathtt{y}_1$, ... $\mathtt{y}_n$ have types $\mathtt{c}_1$, ... $\mathtt{c}_n$, as prescribed by the signature, and return a result of a subtype of $\mathtt{c}_{n+1}$, the return type of the signature. In figure 3 we define verification.

rmFld allows the removal of a field `f` from a class `c`, provided that no method body verification depended on the existence of that field, *i.e.,* all method bodies verify in π', after the removal of the field. Note, that in contrast to rule *ldFld*, we do *not* require that there should be no objects of class `c` on the heap, since this does not affect type soundness [8].

rmMthSig allows the removal of method signature for `m` for class `c`, provided that no method body verification depended on the existence of that method signature, *i.e.,* all method bodies verify in π'.

rmMthBdy allows the removal of method body for `m` for class `c`. Note, that this allows the removal of methods which are currently active on the stack. This is not problematic in our setting with large step operational semantics, which, in some sense, "copies" method bodies upon method call.

rplMthBdy allows replacing the method body for `m` in class `c` by expression `e`, provided that `e` satisfies the method signature.

rmClss allows the removal of class `c`, provided that this class is not the superclass of any other class `c'`, that there exist no objects of that class on the heap, that `c` has no fields or methods defined, and that `c` was not needed in the verification of any method.

Rules *ldClss*, *ldFld*, *ldMthSig* and *ldMthBody* correspond to features **DL1**-**DL4**, and thus, they reflect dynamic linking as in current Java and C# implementations, albeit with finer steps, thus requiring more detailed checks than current implementations. Rule *rmClss* corresponds to current JVM hotswapping.

Rules *rmFld*, *rmMthSig*, *rmMthBody*, and *rmClss* correspond to **DL5**-**DL6**. Because they allow the removal of entities, they make heavy requirements on checking - in effect they require re-checking the complete current program, and, in some cases, they also require scanning the complete heap. The expense of these checks is justified by the fact that runtime removal and replacement are rarely applied, but when they are applied, they are necessitated by the demand to upgrade safely a, possibly safety-critical, system on the fly.

Rules *rmFld*, *rmMthSig*, *rmMthBody*, and *rmClss* combined with rules *ldClss*, *ldFld*, *ldMthSig* and *ldMthBody* have the overall effect of replacement of entities. However, although any replacement can be represented by a sequence of such primitive removal and loading operations, there exist simple replacements, which require a long sequence of such

[7] This requirement is unnecessary in Java and C# implementations, because of dynamic linking is at byte-code level, in which method calls are annotated with the corresponding signature.

[8] This has been pointed out by the second anonymous referee!

$$\frac{}{\pi,\gamma \vdash \mathbf{this} : \gamma(\mathbf{this})\quad \pi,\gamma \vdash \mathtt{y} : \gamma(\mathtt{y})\quad \pi,\gamma \vdash \mathbf{new}\ \mathtt{c} : \mathtt{c}\quad \pi,\gamma \vdash \mathbf{0} : \mathtt{c}}\ \textit{verBasic}$$

$$\frac{\pi,\gamma \vdash \mathtt{e} : \mathtt{c}\quad \mathcal{F}(\pi,\mathtt{c},\mathtt{f}) = \mathtt{c}'}{\pi,\gamma \vdash \mathtt{e.f} : \mathtt{c}'}\ \textit{verFld}$$

$$\frac{\pi,\gamma \vdash \mathtt{e} : \mathtt{c}\quad \mathcal{F}(\pi,\mathtt{c},\mathtt{f}) = \mathtt{c}'\quad \pi,\gamma \vdash \mathtt{e}' : \mathtt{c}''\quad \pi \vdash \mathtt{c}'' \leq \mathtt{c}'}{\pi,\gamma \vdash \mathtt{e.f} := \mathtt{e}' : \mathtt{c}''}\ \textit{verFldAss}$$

$$\frac{\pi,\gamma \vdash \mathtt{e}_i : \mathtt{c}'_i \quad 0 \leq i \leq n\quad \mathcal{MS}(\pi,\mathtt{c}_0,\mathtt{m}) = \mathtt{c}_1,\ldots \mathtt{c}_{n+1} \quad n \geq 0\quad \pi \vdash \mathtt{c}'_i \leq \mathtt{c}_i \quad 1 \leq i \leq n}{\pi,\gamma \vdash \mathtt{e}_0.\mathtt{m}(\mathtt{e}_1,\ldots \mathtt{e}_n) : \mathtt{c}_{n+1}}\ \textit{verMthCll}$$

Figure 3: Verification.

$$\frac{\begin{array}{l}\pi = (\pi_s,\pi_f,\pi_{ms},\pi_{mb})\\ \pi_s(\mathtt{c}) = \mathtt{c}' \neq \epsilon \implies \pi_s(\mathtt{c}') \neq \epsilon\\ \pi_s(\mathtt{Object}) = \mathtt{Object}\\ \pi \vdash \mathtt{c} \leq \mathtt{c}' \text{ and } \pi \vdash \mathtt{c}' \leq \mathtt{c} \implies \mathtt{c} = \mathtt{c}'\\ \pi_f(\mathtt{c},\mathtt{f}) \neq \epsilon \implies \pi_s(\mathtt{c}) \neq \epsilon\\ \pi_{ms}(\mathtt{c},\mathtt{m}) \neq \epsilon \implies \pi_s(\mathtt{c}) \neq \epsilon\\ \pi_{mb}(\mathtt{c},\mathtt{m}) \neq \epsilon \implies \pi_{ms}(\mathtt{c},\mathtt{m}) \neq \epsilon\\ \pi \vdash \mathtt{c}' \leq \mathtt{c},\ \pi_f(\mathtt{c}',\mathtt{f}) \neq \epsilon \implies \pi_f(\mathtt{c},\mathtt{f}) = \epsilon\\ \pi \vdash \mathtt{c}' \leq \mathtt{c},\ \pi_{ms}(\mathtt{c}',\mathtt{m}) \neq \epsilon \implies \pi \vdash \pi_{ms}(\mathtt{c}',\mathtt{m}) \leq \pi_{ms}(\mathtt{c},\mathtt{m})\\ \pi_{mb}(\mathtt{c},\mathtt{m}) = \mathtt{e} \implies\\ \qquad \pi_{ms}(\mathtt{c},\mathtt{m}) = \mathtt{c}_1,\ldots \mathtt{c}_{n+1},\quad n \geq 0\\ \qquad \pi,(\mathbf{this} \mapsto \mathtt{c},\mathtt{y_1} \mapsto \mathtt{c_1},\ldots \mathtt{y_n} \mapsto \mathtt{c_n}) \vdash \mathtt{e} : \mathtt{c}'\\ \qquad \pi \vdash \mathtt{c}' \leq \mathtt{c}_{n+1}\end{array}}{\vdash \pi}$$

Figure 4: Well-formed programs

primitive operations *e.g.,* , replacing the signature of a method which is called by other methods[9]. Therefore, although the choice of the primitive operations is attractive at a theoretical level, it is not necessarily the most natural.

4 Soundness

A program is well formed, judgment $\vdash \pi$ in figure 4, if the class hierarchy forms a tree, no field identifier is re-defined in a subclass and superclass, any method overriding a method of a superclass has signature which is a subsignature of the overridden method, and all methods bodies have a signature, and they verify to that signature. We expect to be able to prove that extension, and hence also evaluation, preserves well-formedness of the program.

A heap and stack conform to a program and environment, judgment $\pi,\gamma \vdash \chi,\sigma$ in figure 5, if the receiver and arguments point to objects which conform to their type declared in the environment. An object conforms to a type $\mathtt{c}$ if it is of class $\mathtt{c}'$ and $\mathtt{c}'$ is a subclass of $\mathtt{c}$, and it contains appropriate values for all fields of the class $\mathtt{c}$.

Soundness of the system guarantees that execution of a well typed expression in an appropriate stack and heap and in context of a well formed program will not get stuck, *i.e.,* will not

[9]This has been pointed out by the second anonymous referee!

$$\frac{\pi \vdash \mathtt{c}' \leq \mathtt{c} \qquad \chi(\iota) = (\,\mathtt{c}',\ \_\,)}{\pi, \chi \vdash \iota \triangleleft \mathtt{c}} \qquad \frac{\chi(\iota) = (\,\mathtt{c},\ fm\,) \qquad \mathcal{F}(\pi, \mathtt{c}, \mathtt{f}) = \mathtt{c}' \implies \pi, \chi \vdash fm(\mathtt{f}) \triangleleft \mathtt{c}'}{\pi, \chi \vdash \iota}$$

$$\frac{}{\pi, \chi \vdash \mathbf{0} \triangleleft \mathtt{c}} \qquad \frac{\chi(\iota) \neq \epsilon \implies \pi, \chi \vdash \iota \qquad \pi, \chi \vdash \sigma(\mathbf{this}) \triangleleft \gamma(\mathbf{this}) \qquad \pi, \chi \vdash \sigma(\mathtt{y}) \triangleleft \gamma(\mathtt{y})}{\pi, \gamma \vdash \chi, \sigma}$$

Figure 5: Conformance.

attempt to access non-existing fields or methods, and if it produces a value, this value is guaranteed to conform with the type of the original expression. Soundness makes no guarantees about link and pointer exceptions. We expect to be able to prove the following Theorem:

Theorem 1 (Soundness) *If*

- $\pi, \gamma \vdash \chi, \sigma,$ *and* $\vdash \pi,$ *and*
- $\pi, \gamma \vdash \mathtt{e} : \mathtt{c}$ *and* $\mathtt{e}, \pi, \sigma, \chi \rightsquigarrow r, \pi', \chi'$

then

- $\vdash \pi',$ *and* $\pi', \gamma \vdash \chi', \sigma,$
- $r \neq \mathtt{StuckExc},$ *and* *if* $r = \kappa$, *then* $\pi', \chi' \vdash \kappa \triangleleft \mathtt{c}$

5 Conclusions and further work

We have developed a formal system which models dynamic linking and re-linking at the language level. at a very fine level. The suggested features support unanticipated software evolution.

Further work includes

- completion of the technical details and proofs,
- an investigation in how far the conditions for program extension in this paper are necessary for preservation of program well-formedness,
- a refinement of the model into a series of models reflecting more and more relevant implementation details (*e.g.*, class layout, object layout, offset calculation), and the restrictions they bring into the concept of program extension,
- an investigation in how far the restriction of Java dynamic linking (eg object creation *after loading, no method removal* together with our current model would lead us to the dynamic linking model as in current Java implementations,
- an investigation into ways of expressing dynamic linking operations at a coarser, or more variable granularity,
- extensions to incorporate ideas for unanticipated object reclassification, [4].

Acknowledgements We are very grateful to Giovanni Lagorio, Mariangiola Dezani and Paola Giannini, and the two anonymous FTfJP referees for their feedback, suggestions, and careful checking and repairs to the formal model. Furthermore, the second anonymous TfJP referee gave insightful suggestions, and ideas on bridging the gap between theory and practical relevance.

References

[1] Davide Ancona, Sonia Fagorzi, and Elena Zucca. A Calculus for Dynamic Linking. In *Italian Conference of Theoretical Computer Science*, 2003.

[2] Gavin Biermann, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing Dynamic Software Updating. In *USE Workshop ETAPS*, April 2003.

[3] Cristiano Calcagno, Peter O'Hearn, and Richard Bornat. Program Logic and Equivalence in the Presence of Gargabe Collection. *Theoretical Computer Science*, 2002.

[4] Ferruccio Damiani and Paola Giannini. Refined Effects for unaticipated object reclassification, Fickle$_3$. In *Italian Conference of Theoretical Computer Science*, 2003.

[5] Mikhail Dimitriev. Hotspot Technology Application for Advanced Profiling. In *ECOOP USE Workhop*, June 2002.

[6] Sophia Drossopoulou and Susan Eisenbach. Manifestations of Java Dynamic Linking. In *ECOOP Workshop on Unanticipated Software Evolution*, 2002.

[7] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is Java Sound? *Theory and Practice of Object Systems*, 5(1), January 1999.

[8] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In *12th European Symposium on Programming*, 2003.

[9] Susan Eisenbach. Dynamic Linking phases in Java and C#, 2002. http://www.doc.ic.ac.uk/ sue/foodexample.html.

[10] James Goslong, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification - Second Edition*. Addison-Wesley, 2000.

[11] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic Software Updating. In *Programming Language Design and Implementation*. ACM, 2001.

[12] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A Formal Specification of Java^{TM} Class Loading. In *OOPSLA'2000*, November 2000.

Appendix

Reachable addresses An address ι *reaches* another address ι' in the context of heap χ, iff for some field `f`, $\chi(\iota)$(`f`)=ι', or $\chi(\iota)$(`f`) reaches ι. An address ι *is reachable* from a stack σ in χ, iff there exists a k, so that $1 \leq k \leq n$ and $\sigma = \phi_1 ... \phi_k ... \phi_n$, and ϕ_k (**this**) reaches ι, or ϕ_k (`y`) reaches ι for some `y`.

Stuck execution and null-pointer exceptions Rules $\mathit{fldNull}$, $\mathit{fldAssNll}$, and $\mathit{callNull}$ describe null pointer exceptions. Rules $\mathit{fldStck}$, $\mathit{fldAssStck}$, and $\mathit{callStck}$ describe stuck execution, due to a missing field or method body. Rules $\mathit{propNull}$, and $\mathit{propStck}$ describe propagation of exceptions.

In particular, as we discussed in section 2, an undefined signature for the particular class, *i.e.*, $\mathcal{MS}(..,..,..) = \epsilon$ indicates that the class has no such method, and then execution is stuck, while a defined signature and undefined method body, *i.e.*, $\mathcal{MS}(..,..,..) \neg \epsilon = \mathcal{MB}(..,..,..)$ indicates a verification error for the method body, and then execution throws a link related exception.

$$\frac{\mathtt{e},\pi,\sigma,\chi \rightsquigarrow \mathbf{0},\pi,\chi'}{\mathtt{e.f},\pi,\sigma,\chi \rightsquigarrow \mathtt{nllPtrExc},\pi,\chi'}\ \textit{fldNull}$$

$$\frac{\mathtt{e},\pi,\sigma,\chi \rightsquigarrow \mathbf{0},\pi',\chi' \quad \mathtt{e}',\pi',\sigma,\chi' \rightsquigarrow \kappa,\pi'',\chi''}{\mathtt{e.f} := \mathtt{e}',\pi,\sigma,\chi \rightsquigarrow \mathtt{nllPtrExc},\pi'',\chi''}\ \textit{fldAssNull}$$

$$\frac{\mathtt{e},\pi,\sigma,\chi \rightsquigarrow \mathbf{0},\pi_1,\chi_1 \quad \mathtt{e}_i,\pi_i,\sigma,\chi_i \rightsquigarrow \kappa_i,\pi_{i+1},\chi_{i+1}\ \ 1 \le i \le n}{\mathtt{e}.m(\mathtt{e}_1,...\mathtt{e}_n),\pi,\sigma,\chi \rightsquigarrow \mathtt{nllPtrExc},\pi',\chi'}\ \textit{callNull}$$

$$\frac{\mathtt{e}',\pi,\sigma,\chi \rightsquigarrow \mathtt{nllPtrExc},\pi,\chi \quad \mathtt{e} \text{ has subexpression } \mathtt{e}'}{\mathtt{e},\pi,\sigma,\chi \rightsquigarrow \mathtt{nllPtrExc},\pi,\chi}\ \textit{propNll}$$

$$\frac{\mathtt{e},\pi,\sigma,\chi \rightsquigarrow \iota,\pi,\chi' \quad \chi'(\iota)(\mathtt{f}) = \epsilon}{\mathtt{e.f},\pi,\sigma,\chi \rightsquigarrow \mathtt{stuckExc},\pi,\chi'}\ \textit{fldStck}$$

$$\frac{\mathtt{e},\pi,\sigma,\chi \rightsquigarrow \iota,\pi',\chi' \quad \mathtt{e}',\pi',\sigma,\chi' \rightsquigarrow \kappa,\pi'',\chi'' \quad \chi''(\iota)(\mathtt{f}) = \epsilon}{\mathtt{e.f} := \mathtt{e}',\pi,\sigma,\chi \rightsquigarrow \mathtt{stuckExc},\pi'',\chi''}\ \textit{fldAssStck}$$

$$\frac{\mathtt{e},\pi,\sigma,\chi \rightsquigarrow \mathbf{0},\pi_1,\chi_1 \quad \mathtt{e}_i,\pi_i,\sigma,\chi_i \rightsquigarrow \kappa_i,\pi_{i+1},\chi_{i+1}\ \ 1 \le i \le n \quad \chi(\iota) = (\ \mathtt{c},\ ...\ ) \quad \mathcal{MS}(\pi,\mathtt{c},\mathtt{m}) = \epsilon}{\mathtt{e}.m(\mathtt{e}_1,...\mathtt{e}_n),\pi,\sigma,\chi \rightsquigarrow \mathtt{stuckExc},\pi',\chi'}\ \textit{callStck}$$

$$\frac{\mathtt{e}',\pi,\sigma,\chi \rightsquigarrow \mathtt{stuckExc},\pi,\chi \quad \mathtt{e} \text{ has subexpression } \mathtt{e}'}{\mathtt{e},\pi,\sigma,\chi \rightsquigarrow \mathtt{stuckExc},\pi,\chi}\ \textit{propStck}$$

Subtypes The judgment $\pi_s \vdash \mathtt{c}' \le \mathtt{c}$ means that $\mathtt{c}'$ is a (possibly indirect) subclass of $\mathtt{c}$ according to π_s, while $\pi_s \vdash \mathtt{c}' \lneq \mathtt{c}$ means that $\mathtt{c}' \neq \mathtt{c}$, and either $\pi_s \vdash \mathtt{c}' \le \mathtt{c}$ or $\pi_s \vdash \mathtt{c} \le \mathtt{c}'$. Both judgments can be extended to full programs, giving judgments $\pi \vdash \mathtt{c}' \le \mathtt{c}$ and $\pi \vdash \mathtt{c}' \lneq \mathtt{c}$:

$$\frac{\pi_s(\mathtt{c}') = \mathtt{c}}{\pi_s \vdash \mathtt{c}' \le \mathtt{c}} \qquad \frac{\pi_s \vdash \mathtt{c}' \le \mathtt{c}'' \quad \pi_s \vdash \mathtt{c}'' \le \mathtt{c}}{\pi_s \vdash \mathtt{c}' \le \mathtt{c}} \qquad \frac{\pi_s \vdash \mathtt{c}' \le \mathtt{c} \quad \pi = (\pi_s,\pi_f,\pi_{ms},\pi_{mb})}{\pi \vdash \mathtt{c}' \le \mathtt{c}}$$

$$\pi_s \vdash \mathtt{c} \le \mathtt{c} \qquad \pi_s \vdash \mathtt{c}' \le \mathtt{c}'$$

$$\frac{\pi_s \vdash \mathtt{c}' \le \mathtt{c} \quad \mathtt{c} \neq \mathtt{c}'}{\pi_s \vdash \mathtt{c}' \lneq \mathtt{c}} \qquad \frac{\pi_s \vdash \mathtt{c} \le \mathtt{c}' \quad \mathtt{c} \neq \mathtt{c}'}{\pi_s \vdash \mathtt{c}' \lneq \mathtt{c}} \qquad \frac{\pi_s \vdash \mathtt{c}' \lneq \mathtt{c} \quad \pi = (\pi_s,\pi_f,\pi_{ms},\pi_{mb})}{\pi \vdash \mathtt{c}' \lneq \mathtt{c}}$$

Subsignatures The judgment $\pi \vdash \mathtt{c}'_1...\mathtt{c}'_n \le \mathtt{c}_1...\mathtt{c}_n$ says that the signature $\mathtt{c}'_1...\mathtt{c}'_n$ may safely override signature $\mathtt{c}_1...\mathtt{c}_n$ in a subclass. The first rule says that any signature may override a non existing definition, while the second rule says that subsignatures are contavariant in argument types and covariant in result types.

$$\frac{}{\pi \vdash \mathtt{c}'_1...\mathtt{c}'_n \le \epsilon} \qquad \frac{\pi \vdash \mathtt{c}_i \le \mathtt{c}'_i \ \ i \in 1..n-1 \quad \pi \vdash \mathtt{c}'_n \le \mathtt{c}_n}{\pi \vdash \mathtt{c}'_1...\mathtt{c}'_n \le \mathtt{c}_1...\mathtt{c}_n}$$