

A Flexible Model for Dynamic Linking in Java and C# *

Sophia Drossopoulou

Department of Computing, Imperial College London

email:sd@doc.ic.ac.uk

Giovanni Lagorio

DISI, University of Genova

email:lagorio@disi.it

Susan Eisenbach

Department of Computing, Imperial College London

email:sue@doc.ic.ac.uk

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, code deployment

*Work partly supported by DART, European Commission research Directorate IST-01-6-1A

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.

Dynamic linking was incorporated into operating systems, *e.g.*, by Multics[41], 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 were the first 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” [32]. The .NET architecture claims to have solved this problem with sophisticated systems of versioning and side-by-side components [33]. 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 [30].

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 [19, 31]. 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.

¹Actually 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.

| Phase | Granularity | | Organization | |
|--------------------|-------------------------------------|------------|---|---|
| | Java | C# | Java | C# |
| load | per class | | after superclass loaded | |
| verify | 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 |

Table 1: Dynamic linking phases, granularity and organization

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 explicitly 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.

This paper presents further work to that presented at ESOP’03 [15] 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.

2 Introduction to the Dynamic Linking Phases

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

```

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);
    }
}

```

Figure 1: Example program

- 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 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 requires some class to extend a further class, it will load the two classes – although [34] 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 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 figure 1 serves to illustrate these points. The concrete example written in both C# and Java, complete with instructions on how to produce the behaviour described in this paper, is available at the following URL:

<http://www.doc.ic.ac.uk/~sue/foodexample.html>

| 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 |

Table 2: Successful execution of the Meal example – assuming Cls,Fld,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 ... | | | |
| 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 3: Execution when ¬Fld

It could be expressed either in Java and in C#, modulo some syntactic modifications and 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 { }
```

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 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

| 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 | | |

Table 4: Execution when $\neg \text{Cls}$

through the \hookrightarrow symbol, *e.g.*, in Java, verification of class Meal requires verification of method eat, which in its turn checks that $\text{Pasta} \leq \text{Penne}$. 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, $\neg \text{Fld}$ 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, $\neg \text{Cls}$ 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, $\neg \text{Sub}$ 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.

| 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 |

Table 5: Execution when \neg Sub

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 figure 2 we give an overview of our terms and judgments, and the figures where they are defined.

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

| | | |
|---|--|---------|
| e | expressions | fig. 3 |
| t | types | fig. 3 |
| ι | addresses | fig. 3 |
| J | offsets | fig. 3 |
| <code>nullPEX</code> | the null-pointer exception | fig. 3 |
| <code>lnkEx</code> | link-related exception, <i>e.g.</i> , 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 |
| ν | code tables | fig. 3 |
| H | heaps | sec. 4 |
| E | environment giving types to receiver/argument | sec. 4 |
| $\square \cdot \square^{exe}$ | execution context | sect. 4 |
| $\square \cdot \square^{off}$ | 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 |
| $W \vdash P' \leq P$ | program P' extends program P in global context W | fig. 7 |
| $P \vdash t' \leq t$ | 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 |
| $P, H \vdash \iota \triangleleft c$ | ι conforms class c , or subclass | fig. 10 |
| $x' \ll x$ | x' is more defined than x | def. 1 |
| $g' \leq g$ | mapping g' extends g | def. 1 |
| $g' \leq g \text{ exc. } A$ | 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 | page 15 |
| $TypFld(P, c, j)$ | the type of the field contained at the offset j of c in P | page 20 |
| $Offst(P, c, t, f)$ | the offset of field f as defined c or some superclass | page 25 |

Figure 2: Overview of terms and judgements

Loading, verification and laying out of classes can be understood as enriching the information in the program, and is represented through the judgement $W \vdash P' \leq P$. 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 judgment $P, e \rightsquigarrow_{W,E} P', e', t$ 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 judgment, $P, e \rightsquigarrow_{W,E} P', e', t$. This judgment requires optional offset calculation for its subexpressions (third, fifth and sixth rule in figure 8). Optional offset calculation either replaces symbolic references by numeric offsets (first and second rule in figure 6), or leaves the symbolic reference unmodified (last rule in figure 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 figure 4), and the operational semantics for member access requires the offset to have been calculated (fourth and fifth rules in figure 4). This describes the Java “lazy” offset calculation.

Our model allows many more executions (which do not correspond to neither Java nor 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 figure 8); thus reflecting Java. On the other hand, any class may be loaded during program extension (third rule in figure 7), and programs may be extended during verification/jit-compilation (fourth rule in figure 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 figure 4), thus describing the C# “lazy” approach. Furthermore, verification/jit-compilation is part of program extension (fifth rule in figure 7), and program extension may take place at any time during execution (first rule in figure 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 figure 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 figure 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 .² In order to describe program extension, we define the concepts of mapping extension (i.e., $g' \leq g$, and $g' \leq g \text{ exc. } A$), 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 | $x = \epsilon$ or $x = x'$. |
| $g' \leq g$ | iff | $\mathcal{D}(g) \subseteq \mathcal{D}(g')$, and $g' _{\mathcal{D}(g)} = g$ |
| $g' \leq g \text{ exc. } A$ | iff | $\mathcal{D}(g') = \mathcal{D}(g) \cup A$, and $g' _{\mathcal{D}(g) \setminus A} = g _{\mathcal{D}(g) \setminus A}$. |
| $g \cdot g'$ | = | a function g'' , with $\mathcal{D}(g'') = \mathcal{D}(g) \cup \mathcal{D}(g')$, and $g''(x) = g'(x)$ if $g'(x) \neq \epsilon$, $g(x)$ 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 $_ \leq _$ are reflexive, and not symmetric. The operation $_ \oplus _$ is commutative, but $_ \otimes _$ is not. When $g' \leq g \text{ exc. } A$ holds, the set A and $\mathcal{D}(g)$, but need not, be disjoint.

Programs reflect the internal representation of code, and are described in figure 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 ($\bar{\kappa} \in \text{FldDescr}$, consisting of field identifiers and types), and method descriptors ($\bar{\mu} \in \text{MthDescr}$, consisting of method identifier, argument type, return type and method body).³ Laid out classes consist of a field layout table

²Notice, that the notation $x' \ll x$ is opposite to the usual definition \sqsubseteq , where $\perp \sqsubseteq v$ for any value v , however, the notation corresponds to the notation for $g' \leq g$, where g' is more defined than g .

³Note that we use the "overbar" to indicate similar entities from different domains, e.g., κ and $\bar{\kappa}$, and *not* output, as in the π -calculus, neither does it indicate a vector, as in Featherweight Java.

| Programs | | |
|-----------------------------|--|---|
| $P \in Prg$ | $= (ClassId \rightarrow (ClassRaw \uplus ClassLaidOut))$ $\times (\mathbb{N} \rightarrow Body)$ | programs |
| $ClassRaw$ | $= ClassId \times FldDescr \times MthDescr$ | |
| $\bar{\kappa} \in FldDescr$ | $= FieldId \rightarrow Typ$ | field descriptions |
| $\bar{\mu} \in MthDescr$ | $= MethId \times Typ \times Typ \rightarrow Exp$ | method descriptions |
| $ClassLaidOut$ | $= ClassId \times FldTbl \times MthTbl \times CdeTbl$ | |
| $\kappa \in FldTbl$ | $= FieldId \times Typ \rightarrow \mathbb{N}^+$ | field layout tables |
| $\mu \in MthTbl$ | $= MethId \times Typ \times Typ \rightarrow \mathbb{N}$ | method layout tables |
| $\nu \in CdeTbl$ | $= \mathbb{N} \rightarrow \mathbb{N}$ | code tables |
| $Body$ | $= (Typ \times Typ \times Exp)$ $\uplus Exp$ | meth. body before jit/verif. meth. body after jit/verif. |
| Global contexts | | |
| $W \in$ | $ClassId \rightarrow ClassRaw$ | |
| Expressions | | |
| $e, e' \in Exp$ | $::= \text{new } c \mid$ $\iota \mid$ $\mathbf{p} \mid$ $e \text{ ma}(e') \mid$ $e \text{ fa} = e' \mid$ $e \text{ fa} \mid$ $\mathbf{this} \mid$ $\mathbf{nullPEx} \mid$ \mathbf{lnkEx} | instance creation address parameter method invocation field assignment field access this reference null-pointer exception linking related exception |
| $t, t' \in Typ$ | $::= c$ | type (class name) |
| $ma \in Ann^M$ | $::= .m[c, t, t'] \mid$ $[j]$ | unresolved method annotation resolved method annotation |
| $fa \in Ann^F$ | $::= .f[c, t] \mid$ $[j]$ | unresolved field annotation resolved field annotation |
| $a \in Ann$ | $::= fa \mid$ ma | field annotation 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 |
| j | $\in \mathbb{N}$ | offsets |

Figure 3: Expressions and programs

($\kappa \in FldTbl$, 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 ($\nu \in CdeTbl$, which maps offsets to addresses of method bodies).⁴

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 $P(i)$ is a shorthand for $P\downarrow_2(i)$. 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 figure 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]`.

Values are addresses, which are natural numbers denoted by ι, ι' *etc.*; the null pointer is $\mathbf{0}$.⁵ `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, 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 ι 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 $\mathbf{0}$. Heaps therefore have the form:

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

We implicitly require the sets \mathbb{N} and $ClassId$ to be disjoint. The lookup $H(\iota)$ returns the contents at ι in H . If $H(\iota) = c \in ClassId$ then ι points to an object of class c . The fields of that object are stored at some offset, j , from ι . An address ι is *fresh* in heap H iff $\forall j : H(\iota + j) = \epsilon$.

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

⁴Appendix A.1 contains an example clarifying descriptions and layout tables in the presence of method inheritance and field hiding.

⁵Adding 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**.

| | |
|--|--|
| $\frac{W \vdash P' \leq P}{P, H, e \rightsquigarrow_W P', H, e}$ | $\frac{}{P, H, e \rightsquigarrow_W P, H, \text{lnkEx}}$ |
| $\frac{FdOffs(P, c) = \{j_1, \dots, j_n\}, \quad \iota \text{ fresh in } H}{P, H, \text{new } c \rightsquigarrow_W P, H[\iota \mapsto c, \iota + j_1 \mapsto \mathbf{0}, \dots, \iota + j_n \mapsto \mathbf{0}], \iota}$ | |
| $\frac{H(\iota) = c \quad P(c) = \langle _ , _ , _ , \nu \rangle \quad P(\nu(j)) = e}{P, H, \iota[j](\iota') \rightsquigarrow_W P, H, e[\iota/\text{this}, \iota'/p]}$ | $\frac{\iota \neq 0}{P, H, \iota[j] \rightsquigarrow_W P, H, H(\iota + j) \quad P, H, \iota[j] = \iota' \rightsquigarrow_W P, H[\iota + j \mapsto \iota'], \iota'}$ |
| $\frac{}{P, H, \mathbf{0}[j] \rightsquigarrow_W P, H, \text{nllPEX} \quad P, H, \mathbf{0}[j] = \iota \rightsquigarrow_W P, H, \text{nllPEX} \quad P, H, \mathbf{0}[j](\iota) \rightsquigarrow_W P, H, \text{nllPEX}}$ | $\frac{P, H, e \rightsquigarrow_W P', H', e' \quad P, H, \square e \square^{exe} \rightsquigarrow_W P', H', \square e' \square^{exe}}{z = \text{nllPEX}, \text{ or } z = \text{lnkEx} \quad P, H, \square z \square^{exe} \rightsquigarrow_W P, H, z}$ |
| $\frac{a \rightsquigarrow_P a'}{P, H, \square a \square^{off} \rightsquigarrow_W P, H, \square a' \square^{off}}$ | |

Figure 4: Execution and evaluation

| | | |
|--------------|--------------|---------------------------------------|
| $H_0(2)$ | = Penne | start Penne object |
| $H_0(3)$ | = 55 | field <code>int</code> cal from Pasta |
| $H_0(4)$ | = Food | start Food object |
| $H_0(\iota)$ | = ϵ | for all other ι 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 [11], heaps are modelled at a lower level than in verifier studies [37, 23, 34], 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.⁶

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

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

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

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

⁶On 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 modeling of that aspect would not have promoted the study of dynamic linking.

| | |
|---|---|
| $\frac{P(c_1) \downarrow_1 = c_2}{P \vdash c_1 \leq c_1}$ $P \vdash c_1 \leq c_2$ | $\frac{P \vdash c_1 \leq c_2}{P \vdash c_2 \leq c_3}$ $P \vdash c_1 \leq c_3$ |
|---|---|

Figure 5: Subtypes

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

Figure 6: Optional offset calculation

Creation of a new object of class c , $\text{new } c$, allocates fresh addresses for the fields of c at the corresponding offsets, initializing them with $\mathbf{0}$. It requires the auxiliary function $FdOffs(P, c)$ which collects the field offsets from all superclasses:⁷

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

Method invocation, $\iota[j](\iota')$, looks up the method body e in $H(\iota)$, the dynamic class of the receiver ι , using the offset $\nu(j)$, and executes that body after replacing `this` by the actual receiver ι , and the parameter `p` by the argument ι' . Therefore, evaluation only applies to expressions which do *not* contain `this`, or `p`. The format of the invocation $\iota[j](\iota')$ (rather than $\iota.m[c, t_r, t_p](\iota')$) means that the offset has been calculated. The requirement $P(c) = \langle _ , _ , _ , \nu \rangle$ (rather than $P(c) = \langle _ , _ , _ \rangle$) means that the class c has been laid out. The requirement that $P(\nu(j)) = e$ (rather than $P(\nu(j)) = \langle _ , _ , _ \rangle$) 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 invocation and field access for $\mathbf{0}$ throw a `nllPEX`, as described in the sixth rule of the table.

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:

$$\begin{aligned} \square \cdot \square^{exe} &::= \square \cdot \square^{exe} \text{ ma}(e) \mid \iota \text{ ma}(\square \cdot \square^{exe}) \mid \\ &\square \cdot \square^{exe} \text{ fa} = e \mid \iota \text{ fa} = \square \cdot \square^{exe} \mid \square \cdot \square^{exe} \text{ fa} \end{aligned}$$

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 figure 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 figure 6 says that offsets for

⁷Note, that the function $FdOffs(P, c)$ is well-defined, even if the program P should contain cycles in the class hierarchy.

| | | |
|---|---|--|
| $\frac{}{W \vdash P \leq P}$ | $\frac{W \vdash P' \leq P'' \quad W \vdash P'' \leq P}{W \vdash P' \leq P}$ | $\frac{W(c) = \langle c_s, _ , _ \rangle \quad P(c_s) \neq \epsilon}{W \vdash P \oplus [c \mapsto W(c)] \leq P}$ |
| $P(c) = \langle c_s, \bar{\kappa}, \bar{\mu} \rangle, P(c_s) = \langle _ , \kappa_s, \mu_s, \nu_s \rangle$ (a) | | |
| κ injective, $\mathcal{D}(\kappa) = \bar{\kappa} \downarrow$, $\mathcal{R}(\kappa) \cap \text{FdOffs}(P, c_s) = \emptyset$ (b) | | |
| μ injective, $\mu \leq \mu_s$, $\mathcal{D}(\mu) = \mathcal{D}(\mu_s) \cup \mathcal{D}(\bar{\mu})$ (c) | | |
| $\nu \leq \nu_s$ exc. $\mu(\mathcal{D}(\bar{\mu}))$ (d) | | |
| $\bar{\mu}(m, t, t') = e \implies$ (e) | | |
| $\exists j, 1 \leq j \leq n : \iota_j = \nu(\mu(m, t, t')), t_j = t, t'_j = t', e_j = e$ | | |
| $\frac{}{W \vdash P \otimes [c \mapsto \langle c_s, \kappa, \mu, \nu \rangle] \oplus [\iota_1 \mapsto \langle t_1, t'_1, e_1 \rangle, \dots, \iota_n \mapsto \langle t_n, t'_n, e_n \rangle] \leq P}$ | | |
| $P(c) = \langle _ , _ , \mu, \nu \rangle, \iota = \nu(\mu(_ , t_r, t_p)), P(\iota) = \langle t_r, t_p, e_0 \rangle$ (a) | | |
| $\iota \in \mathcal{R}(P(c') \downarrow_d) \implies P \vdash c' \leq c$ (b) | | |
| $P, e_0 \rightsquigarrow_W, \{\text{this} \mapsto c, p \mapsto t_p\} P'', e, t$ (c) | | |
| $P'', t, t_r \rightsquigarrow_W P'$ (d) | | |
| $\frac{}{W \vdash P' \otimes [\iota \mapsto e] \leq P}$ | | |

Figure 7: Program extension

methods are looked up in the method layout table of the particular class, under the given argument and return type, 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.⁸ The last rule allows optional offset calculation to leave a unmodified. The last rule in figure 4 allows offset calculation to happen during execution, as in Java. For this, we have defined offset calculation contexts as:

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

Optional offset calculation also happens during verification/jit-compilation (figure 8). If one of the two first rules from figure 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

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

c.f. figure 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 [11], but is needed when formulating and proving equivalence of strategies.

In more detail, $W \vdash P' \leq P$ is defined as follows:

- The first and second rules state that $W \vdash P' \leq P$, if P' is in the reflexive and

⁸An example of this is shown in appendix A.

transitive closure of the extension relation $\dots \vdash \dots \leq P$.

- 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 $\langle c_s, \kappa, \mu, \nu \rangle$, and the unverified method bodies from the raw version of c are given fresh addresses ι_1, \dots, ι_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 , extends⁹ 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. ν , 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 ν_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.
- 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 ι .
 - b. The class c is the most general superclass of all classes c' which may contain the address ι in their virtual table.
 - 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. Figure 4).

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

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

defined in figure 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

⁹For simplicity, we do not model the C# `new` modifier that introduces a method with the same signature as an inherited one without overriding it.

| | |
|--|---|
| $\frac{}{P, \mathbf{this} \rightsquigarrow_{W,E} P, \mathbf{this}, E(\mathbf{this})}$ $\frac{}{P, \mathbf{p} \rightsquigarrow_{W,E} P, \mathbf{p}, E(\mathbf{p})}$ $P, e_1 \rightsquigarrow_{W,E} P_1, e'_1, t_1$ $P_1, e_2 \rightsquigarrow_{W,E} P_2, e'_2, t_2$ $P_2, t_1, c \rightsquigarrow_W P_3$ $P_3, t_2, t_p \rightsquigarrow_W P'$ $\frac{}{.m[c, t_r, t_p] \rightsquigarrow_{P'} ma}$ <hr style="border: 0.5px solid black;"/> $P, e_1.m[c, t_r, t_p](e_2) \rightsquigarrow_{W,E} P', e'_1 ma(e'_2), t_r$ $P, e_1 \rightsquigarrow_{W,E} P_1, e'_1, t_1$ $P_1, e_2 \rightsquigarrow_{W,E} P_2, e'_2, t_2$ $P_2, t_1, c \rightsquigarrow_W P_3$ $P_3, t_2, t_f \rightsquigarrow_W P'$ $\frac{}{.f[c, t_f] \rightsquigarrow_{P'} fa}$ <hr style="border: 0.5px solid black;"/> $P, e_1.f[c, t_f] = e_2 \rightsquigarrow_{W,E} P', e'_1 fa = e'_2, t_f$ $\frac{}{W \vdash P' \leq P}$ $\frac{}{P' \vdash t' \leq t}$ <hr style="border: 0.5px solid black;"/> $P, t', t \rightsquigarrow_W P'$ | $\frac{}{P, c, c \rightsquigarrow_W P'}$ <hr style="border: 0.5px solid black;"/> $P, \mathbf{new} c \rightsquigarrow_{W,E} P', \mathbf{new} c, c$ $P, 0 \rightsquigarrow_{W,E} P', 0, c$ $\frac{}{W \vdash P'' \leq P}$ $\frac{}{P'', e \rightsquigarrow_{W,E} P', e', t}$ <hr style="border: 0.5px solid black;"/> $P, e \rightsquigarrow_{W,E} P', e', t$ $P, e \rightsquigarrow_{W,E} P_1, e', t_e$ $P_1, t_e, c \rightsquigarrow_W P'$ $\frac{}{.f[c, t_f] \rightsquigarrow_{P'} fa}$ <hr style="border: 0.5px solid black;"/> $P, e.f[c, t_f] \rightsquigarrow_{W,E} P', e' fa, t_f$ <hr style="border: 0.5px solid black;"/> $P, t, t \rightsquigarrow_W P$ |
|--|---|

Figure 8: Verification and Jit-compilation

E which maps `this` and the parameter `p` to types, *i.e.*, $E : \{ \mathbf{this}, \mathbf{p} \} \rightarrow Typ$, 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 $\mathbf{0}$ 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 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 fourth 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

$$P, t_1, t_2 \rightsquigarrow_W P'$$

which is also given in figure 8. Notice, that the last rule in this figure allows any identifier to be a subtype of itself even if the identifier has not been loaded - this follows the “optimistic” Java approach.

$$\begin{array}{l}
P(c) = \langle c_s, _ , _ \rangle \implies P(c_s) \neq \epsilon \quad (a) \\
P(c) = \langle c_s, _ , _ , _ \rangle \implies P(c_s) = \langle _ , _ , _ , _ \rangle \quad (b) \\
P(c) = \langle _ , \kappa, \mu, \nu \rangle \implies \begin{cases} \kappa, \mu \text{ injective} \\ \mathcal{R}(\mu) = \mathcal{D}(\nu) \\ \mathcal{R}(\nu) \subseteq \mathcal{D}(P \downarrow_2) \end{cases} \quad (c) \\
\left. \begin{array}{l} c \neq c_s, \\ P \vdash c \leq c_s, \\ P(c) = \langle _ , \kappa, \mu, _ \rangle, \\ P(c_s) = \langle _ , \kappa_s, \mu_s, _ \rangle \end{array} \right\} \implies \begin{cases} \mathcal{R}(\kappa_s) \cap \mathcal{R}(\kappa) = \emptyset \\ \mu \leq \mu_s \end{cases} \quad (d) \\
\frac{P(c) = \langle _ , _ , \mu, \nu \rangle, P(\nu(\mu(_ , t_r, t_p))) = e \implies \\ \exists e_0, t : P, e_0 \rightsquigarrow_{\emptyset, \{\text{this} \mapsto c, \text{p} \mapsto t_p\}} P, e, t, \text{ and } P \vdash t \leq t_r}{\vdash P} \quad (e)
\end{array}$$

Figure 9: Well-formed programs

5 Soundness

Well-Formed Programs The judgment $\vdash P$, describing well-formed programs, is defined in figure 9 and requires the following:

- a. The superclass of any raw class from P is defined in P ;
- 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 [15], 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 Figure 10 defines conformance. The judgment $P, H \vdash \iota$ expresses that the object stored at ι conforms to its class, c , as stored in $H(\iota)$. 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 j in a particular class c is described through the auxiliary function $TypFld(P, c, j)$ ¹⁰:

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

The judgment $P \vdash H$ 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 judgment $P, H \vdash e : t$, from figure 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. $P, e \rightsquigarrow_{W, E} P', e', t \implies W \vdash P' \leq P.$
2. $P, H, e \rightsquigarrow_W P', H', e' \implies W \vdash P' \leq P.$

¹⁰Note that $TypFld(P, c, j)$ is well-defined even the class hierarchy in P should contain cycles.

$$\begin{array}{c}
\frac{P \vdash c' \leq c}{P, H \vdash \iota \triangleleft c} \quad \frac{H(\iota) = c'}{P, H \vdash \mathbf{0} \triangleleft c} \\
\\
\frac{H(\iota) = c \quad \forall j: \text{TypFld}(P, c, j) = t \implies P, H \vdash H(\iota + j) \triangleleft t \quad 1 \leq j \leq \max(\text{FdOffs}(P, c)) \implies H(\iota + j) \notin \text{ClassId}}{P, H \vdash \iota} \\
\\
\frac{H(\iota) \in \text{ClassId} \implies P, H \vdash \iota}{P \vdash H}
\end{array}$$

Figure 10: Conformance

$$3. P, t', t \rightsquigarrow_W P \implies P \vdash t' \leq t.$$

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

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

$$\left. \begin{array}{l} P, e_0 \rightsquigarrow_{W, \{\text{this} \rightarrow c, \text{p} \rightarrow t_p\}} P, e, t \text{ and} \\ P \vdash c' \leq c \end{array} \right\} \implies \left\{ \begin{array}{l} P, e_0 \rightsquigarrow_{W, \{\text{this} \rightarrow c', \text{p} \rightarrow t_p\}} P, e, t' \\ P \vdash t' \leq t \end{array} \right.$$

Proof Sketch: By structural induction on $P, e_0 \rightsquigarrow_{W, \{\text{this} \rightarrow c, \text{p} \rightarrow t_p\}} P, e, t$.

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 [13], used in [9, 11], and explored in our model of binary compatibility [14]. Notice that such properties do not always hold for source code, *c.f.* [6] for counterexamples.

Lemma 3 *If $W \vdash P' \leq P$, then*

1. $P'(c) \downarrow_1 \ll P(c) \downarrow_1$
2. $\text{TypFld}(P', c, j) \ll \text{TypFld}(P, c, j)$.
3. $\text{FdOffs}(P', c) \ll \text{FdOffs}(P, c)$.
4. $P \vdash t_1 \leq t_2 \implies P' \vdash t_1 \leq t_2$.

| | | |
|---|---|---|
| $\frac{}{P, H \vdash \mathbf{0} : c}$ $\frac{}{P, H \vdash \text{new } c : c}$ | $\frac{H(\iota) = c' \quad P \vdash c' \leq c}{P, H \vdash \iota : c}$ | $\frac{P, H \vdash e : c' \quad P \vdash c' \leq c}{P, H \vdash e.f[c, t] : t}$ |
| $\frac{P, H \vdash e : c \quad \text{TypFld}(P, c, j) = t}{P, H \vdash e[j] : t}$ | $\frac{P, H \vdash efa : t \quad P, H \vdash e' : t' \quad P \vdash t' \leq t}{P, H \vdash efa = e' : t}$ | |
| $\frac{P, H \vdash e_1 : c_1 \quad P, H \vdash e_2 : t_2 \quad P \vdash c_1 \leq c \quad P \vdash t_2 \leq t_p}{P, H \vdash e_1.m[c, t_r, t_p](e_2) : t_r}$ | $\frac{P, H \vdash e_1 : c_1 \quad P, H \vdash e_2 : t_2 \quad P \vdash t_2 \leq t_p \quad P(c_1) = \langle _ , _ , \mu, _ \rangle \quad \mu(m, t_r, t_p) = j}{P, H \vdash e_1[j](e_2) : t_r}$ | |

Figure 11: Types of runtime expressions

5. $P, H \vdash \iota \implies P', H \vdash \iota$.
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. $P, e \rightsquigarrow_{W,E} P, e', t \implies P', e \rightsquigarrow_{W,E} P', e', t$.¹¹

Proof Sketch: The assertions 1-3 are proven by structural induction over the judgment $W \vdash P' \leq P$. The remaining assertions are proven by structural induction over the judgments of each of the assertions, *i.e.*, over $P \vdash t_1 \leq t_2$, or $P, H \vdash \iota$ *etc.*

Lemma 4 $W \vdash P' \leq P$ and $\vdash P \implies \vdash P'$.

Proof Sketch: : by structural induction over the derivation of $W \vdash P' \leq P$, 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.

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:¹²

¹¹Notice, 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.

¹²Notice, 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, which would give a type to `this` and to `p`.

Lemma 5 For any $P, e, e', c, t, \iota, \iota', t_p$:

$$\left. \begin{array}{l} \vdash P \\ P, e' \rightsquigarrow_{\emptyset, \{\text{this} \mapsto c, \text{p} \mapsto t_p\}} P, e, t \\ P, H \vdash \iota \triangleleft c \\ P, H \vdash \iota' \triangleleft t_p \end{array} \right\} \Longrightarrow \left\{ \begin{array}{l} \exists t' : \\ P, H \vdash e[\iota/\text{this}, \iota'/\text{p}] : t', \\ P \vdash t' \leq t \end{array} \right.$$

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

Proof Sketch: Use structural induction over the judgment $P, e \rightsquigarrow_{\emptyset, \{\text{this} \mapsto c, \text{p} \mapsto t_p\}} P, e', t$. The base cases are straightforward. Let us consider the inductive case where the last rule applied was the sixth rule from figure 8. This implies that e is a field access, *i.e.*, has the form $e_1.f[c, t_f]$. We distinguish two cases: 1st Case: the optional offset calculation for $.f[c, t]$ replaces the symbolic annotation by a numeric offset, and 2nd Case: the optional offset calculation leaves the symbolic annotation unmodified. In the 1st Case, we use the fact that $.f[c, t] \rightsquigarrow_P [j]$, and $\vdash P$ imply that $\text{TypFld}(P, c, j) = t$; the rest follows by application of the induction hypothesis, and the type rules from figure 11. In the 2nd Case, the hypothesis follows directly from application of the induction hypothesis, and the type rules from figure 11.

The other inductive cases are analogous.

End of Proof Sketch

Execution of a well-typed expression e does not overwrite objects, rather it always creates 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 [13, 40, 11]. 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. $H(\iota) = c \implies H'(\iota) = c$.
2. $H'(\iota) = c \implies H(\iota) = c$ or ι free in H .
3. $P, H \vdash e'' : t'' \implies P', H' \vdash e'' : t''$.

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.*, [13]; 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 :

$$\left. \begin{array}{l} P \vdash H \\ \vdash P \\ P, H \vdash e : t \\ P, H, e \rightsquigarrow_W P', H', e' \end{array} \right\} \implies \left\{ \begin{array}{l} P' \vdash H' \\ \text{if } e' \text{ does not contain exceptions, then} \\ \exists t' : P', H' \vdash e' : t', P' \vdash t' \leq t. \end{array} \right.$$

Proof Sketch: by structural induction over the typing of e .

6 Equivalence of execution strategies

In this section we show that all execution strategies are equivalent, *i.e.*, that nondeterminism 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 $e, P, P', P'', H, H', H'', z$, and $z', z' \in \mathbb{N} \cup \{\text{nilPEX}\}$:

$$\left. \begin{array}{l} P, H, e \rightsquigarrow_W^* P', H', z \\ P, H, e \rightsquigarrow_W^* P'', H'', z' \end{array} \right\} \implies z = z', H' = H'' \text{ up to renaming of addresses}$$

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 from all 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:

- $P(c) = \langle _, _, \mu, \nu \rangle \implies \mathcal{R}(\mu) \subseteq \mathcal{D}(\nu)$,
- $P(c, m, t_r, t_p) = e \implies e$ 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.

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

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

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 $Offst(P, c, t, f)$ which returns the offset of field f of type t as defined in class c or some superclass:

$$Offst(P, c, t, f) = \begin{cases} \epsilon & \text{if } P(c) = \langle _, _, _ \rangle, \text{ or} \\ & P(c) = \langle _, _, _ \rangle, \text{ and} \\ & \forall c', P \vdash c \leq c', P(c) = \langle _, \kappa, _, _ \rangle : \kappa(f, t) = \epsilon \\ j & \text{if } P(c) = \langle _, \kappa, _, _ \rangle, \kappa(f, t) = j \\ j & \text{otherwise, and where } j = Offst(P, P(c) \downarrow_1, t, f) \end{cases}$$

Note that $Offst(P, c, t, f)$ 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:

$$\sigma : \mathbb{N} \rightarrow \mathbb{N}, \quad \text{where } \sigma(0) = 0,$$

which renames addresses across two heaps, preserving the address 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. $P, P', H, H', \sigma \vdash \iota \sim \iota'$ means that addresses ι and ι' point to equivalent objects, *i.e.*, to objects of 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 .¹³
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) $z \equiv \text{n11PEX} \equiv z'$, or
 - (b) $z, z' \in \mathbb{N}$, and $\sigma(z) = z'$.
2. $P, P', H, H', \sigma \vdash \iota \sim \iota'$ iff $\exists c$, so that
 - (a) $H(\iota) = c$, and $H'(\iota') = c$,
 - (b) $P(c) = \langle _ , _ , _ , _ \rangle$, $P'(c) = \langle _ , _ , _ , _ \rangle$,
 - (c) $\forall f, t_f, j : j = \text{Offst}(P, c, t_f, f) \neq \epsilon \implies \sigma(\iota + j) = \iota' + \text{Offst}(P', c, t_f, f)$,
 - (d) $\forall f, t_f, j' : j' = \text{Offst}(P', c, t_f, f) \neq \epsilon \implies \sigma(\iota + \text{Offst}(P, c, t_f, f)) = \iota' + j'$.
3. $P, P', \sigma \vdash H \sim H'$ iff $\sigma(\iota) = \iota' \implies P, P', H, H', \sigma \vdash \iota \sim \iota'$.
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) $fa \equiv _ . f[c, t_f]$, $fa' \equiv [j]$, and $P'(c) \downarrow_2(f, t_f) = j$.
 - (c) $fa \equiv [j]$, $fa' \equiv _ . f[c, t_f]$, and $P(c) \downarrow_2(f, t_f) = j$.¹⁶
 - (d) $fa \equiv [j]$, $fa' \equiv [j']$, and $\exists f : P(c) \downarrow_2(f, t_f) = j$, $P'(c) \downarrow_2(f, t_f) = j'$.
5. $P, P', c, t_r, t_p \vdash ma \sim ma'$ iff one of the following cases holds:

¹³We are using the vague term “can be considered to have type” to express that the expressions are not necessarily well-types in the sense of fig. 11; the judgment $P, P', H, H', E, \sigma \vdash e \sim e' : t$ 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.

¹⁴The last two requirements ensure that c has been laid out both in P and P' , or in none, or that it has no fields in either P or P' . On the other hand if it has a field in P , then it must have a corresponding field in P' , and the opposite.

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

¹⁶This case is the dual to the previous one.

- (a) $ma \equiv ma', ma \equiv \cdot\text{[_ , - , _]}$.¹⁷
- (b) $ma \equiv \cdot m[c, t_r, t_p], ma' \equiv [j]$, and $P'(c)\downarrow_3(m, t_r, t_p) = j$.
- (c) $ma \equiv [j], ma' \equiv \cdot m[c, t_r, t_p]$, and $P(c)\downarrow_3(m, t_r, t_p) = j$.¹⁸
- (d) $ma \equiv [j], ma' \equiv [j'], \exists m, P(c)\downarrow_3(m, t_r, t_p) = j, P'(c)\downarrow_3(m, t_r, t_p) = j'$.
6. $P, P', H, H', E, \sigma \vdash e' \sim e : t$ iff one of the following cases holds:
- (a) $e \equiv \text{this} \equiv e'$, and $t = E(\text{this})$.
- (b) $e \equiv \iota$, and $e' \equiv \iota'$, and $\sigma(\iota) = \iota'$, and $t = H(\iota)$.
- (c) $e \equiv p \equiv e'$, and $t = E(p)$.
- (d) $e \equiv \text{new } c \equiv e'$, 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) $e \equiv e_1 fa = e_2$, 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. $P, P', c, t_p \vdash ma \sim ma'$,
 - 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' .¹⁹
8. $\vdash P \sim P'$ iff the following conditions hold:
- (a) $P(c) = \langle _ , _ , _ \rangle, P'(c) = \langle _ , _ , _ \rangle \implies P(c) = P'(c)$.
- (b) $P(c) = \langle c', \bar{\kappa}, \bar{\mu} \rangle, P'(c) = \langle c'', \kappa, \mu, \nu \rangle \implies$
- 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) $P'(c) = \langle c', \bar{\kappa}, \bar{\mu} \rangle, P(c) = \langle c'', \kappa, \mu, \nu \rangle \implies \dots$ dual to earlier case.
- (d) $P(c) = \langle c'', \kappa, \mu, \nu \rangle, P'(c) = \langle c''', \kappa', \mu', \nu' \rangle \implies$
- i. $c'' = c'''$,
 - ii. $\mathcal{D}(\kappa) = \mathcal{D}(\kappa')$,
 - iii. $\mathcal{D}(\mu) = \mathcal{D}(\mu')$,

¹⁷As 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 .

¹⁸This case is the dual to the previous.

¹⁹Since 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' .

$$(e) \quad P(c, m, t_r, t_p) = e, \text{ and } P'(c, m, t_r, t_p) = e' \implies P, P', E \vdash e \sim e' \\ \text{for } E \equiv (\text{this} \mapsto c, \mathbf{p} \mapsto t_p).$$

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) $P(c) = \langle c', \kappa, \mu, \nu \rangle \implies W(c) = \langle c', \bar{\kappa}, \bar{\mu} \rangle$, and
 - i. $\mathcal{D}(\kappa) = \bar{\kappa} \downarrow$,
 - ii. $\mathcal{D}(\mu) = \mathcal{D}(\bar{\mu}) \cup \mathcal{D}(P(c') \downarrow \mathfrak{g})$,
- (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 $E \equiv (\text{this} \mapsto c, \mathbf{p} \mapsto t_p)$.

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

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

where the notation $\overset{\text{ext}_W^*}{\rightsquigarrow}$ indicates that only program extension steps have been applied, and in the global context W .

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

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

where the notation $\overset{\text{core}^*}{\rightsquigarrow}$ 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:

$$(Prop_1a) \quad \left. \begin{array}{l} P, H, e \overset{\text{core}^*}{\rightsquigarrow} P', H', e', \\ P, H', e' \overset{\text{ext}_W^*}{\rightsquigarrow} P', H', e'. \end{array} \right\} \implies \left\{ \begin{array}{l} P, H, e \overset{\text{ext}_W^*}{\rightsquigarrow} P', H, e, \\ P', H, e \overset{\text{core}^*}{\rightsquigarrow} P', H', e'. \end{array} \right.$$

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.*,

$$(Prop_1) \quad P, H, e \rightsquigarrow_W^* P', H', e' \implies \left\{ \begin{array}{l} P, H, e \overset{\text{ext}_W^*}{\rightsquigarrow} P', H, e, \\ P', H, e \overset{\text{core}^*}{\rightsquigarrow} P', H', e'. \end{array} \right.$$

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

$$(Prop_2a) \quad \left. \begin{array}{l} .f[c, t_f] \rightsquigarrow_P fa \implies P, P, c, t_f \vdash .f[c, t_f] \sim fa. \\ .m[c, t_r, t_p] \rightsquigarrow_P ma \implies P, P, c, t_r, t_p \vdash .m[c, t_r, t_p] \sim ma. \end{array} \right.$$

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

$$(Prop_2b) \quad \left. \begin{array}{l} P, P', H, H', E, \sigma \vdash e \sim e' : t, \\ W \vdash P'' \leq P \end{array} \right\} \implies P'', P', H, H', E, \sigma \vdash e \sim e' : t.$$

Using $(Prop_2a)$ and $(Prop_2b)$ we can prove that jit-compilation/verification creates an equivalent expression, *i.e.*,

$$(Prop_2c) \quad P, e \rightsquigarrow_{W, E} P', e', t \implies P, P', E \vdash e \sim e'.$$

Using $(Prop_2b)$, $(Prop_2c)$, and structural induction on $W \vdash P' \leq P$, we then prove that program extension preserves agreement with the global context, and creates an equivalent program, *i.e.*,

$$(Prop_2d) \quad \left. \begin{array}{l} W \vdash P, \\ W \vdash P' \leq P \end{array} \right\} \implies \left\{ \begin{array}{l} W \vdash P', \\ \vdash P \sim P'. \end{array} \right.$$

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

$$(Prop_2e) \quad \left. \begin{array}{l} W \vdash P, W \vdash P', W \vdash P'', \\ \vdash P \sim P'', \vdash P'' \sim P' \end{array} \right\} \implies \vdash P \sim P'.$$

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.*,

$$(Prop_2) \quad \left. \begin{array}{l} W \vdash P, W \vdash P', \\ \vdash P \sim P', \\ P, H, e \xrightarrow{W}^* P'', H, e, \\ P', H', e' \xrightarrow{W}^* P''', H', e' \end{array} \right\} \implies \vdash P'' \sim P''.$$

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

$$(Prop_3a) \quad \left. \begin{array}{l} \sigma \vdash P, H \sim P', H', \\ P, P', H, H', E, \sigma \vdash e \sim e' : t, \\ P, H, e \xrightarrow{W} P'', H'', e'', \\ P', H', e' \xrightarrow{W} P''', H''', e''' \\ e'' \neq \text{lnkEx} \neq e''' \end{array} \right\} \implies \left\{ \begin{array}{l} \exists \sigma', \sigma' \leq \sigma : \\ \sigma' \vdash P, H \sim P', H'', \\ (P, P', H, H', E, \sigma' \vdash e'' \sim e' : t, \\ \text{or} \\ P, P', H, H', E, \sigma' \vdash e''' \sim e : t, \\ \text{or} \\ P, P', H, H', E, \sigma' \vdash e''' \sim e'' : t). \end{array} \right.$$

Note, that $(Prop_3a)$ allows for three possibilities: the two new expressions may be equivalent (if both steps are offset calculations²⁰, or both are *not* offset calculations), or one of the new expressions is equivalent with one of the old ones (if one step is an offset calculation²¹, 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).

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, *i.e.*,

$$(Prop_3) \quad \left. \begin{array}{l} \sigma \vdash P, H \sim P', H', \\ P, P', H, H', E, \sigma \vdash e \sim e' : t, \\ P, H, e \xrightarrow{W}^* P'', H'', z, \\ P', H', e' \xrightarrow{W}^* P''', H''', z' \end{array} \right\} \implies \left\{ \begin{array}{l} \exists \sigma', \sigma' \leq \sigma : \\ \sigma' \vdash P'', H'' \sim P''', H''', \\ \sigma' \vdash z \sim z'. \end{array} \right.$$

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

$$(Thm_2) \quad \left. \begin{array}{l} W \vdash P, \\ P, H, e \xrightarrow{W}^* P', H', z, \\ P, H, e \xrightarrow{W}^* P'', H'', z', \end{array} \right\} \implies \left\{ \begin{array}{l} \exists \sigma : \\ \sigma \vdash P', H' \sim P'', H'', \\ \sigma \vdash z \sim z'. \end{array} \right.$$

(Thm_2) is a consequence of properties $(Prop_1)$, $(Prop_2)$, and $(Prop_3)$.

²⁰or propagations of offset calculation to the context

²¹or propagation of offset calculation to the context

End of Proof Sketch

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 $z \in \mathbb{N} \cup \{\text{nullPEX}\}$:*

$$\begin{array}{l} P, H, e \rightsquigarrow_W^* P', H', z \\ W \upharpoonright_{\text{def}(P')} = W' \upharpoonright_{\text{def}(P')} \end{array} \implies P, H, e \rightsquigarrow_{W'}^* P', H', z$$

We could probably have replaced the requirement $W \upharpoonright_{\text{def}(P')} = W' \upharpoonright_{\text{def}(P')}$, 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 rather new, very powerful language feature with complex semantics, and which 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 [11]. 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;²²
- 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 significantly simplify the model.

Non-determinism seems to have been in the Java designers’ minds: the specification [26], sect. 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

²²In [11] we distinguished between these, thus keeping a natural distinction, but having a more complex model.

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 [37, 24]. Dynamic loading in Java is formalized in [28], while problems with security in the presence of multiple loaders are reported in [35], a solution presented in [29], which is found flawed and improved upon in [34]. Computation does not preserve types but is type sound. Java's multiple loaders are modeled in [43] 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 [25].

Interest in linking as part of the program lifecycle was kindled through [8]. A collection of examples that demonstrate small details of the dynamic linking process in Java can be found in [12]. Separate compilation for Java is discussed in [3]. Module interconnection languages, and mixins [42, 4, 20, 16, 21] 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 [1]; 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 [7]. A computational interpretation for Hilbert's choice operator is suggested in [2]; 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, [22], and [30], sect. 13, *i.e.*, changes which do not introduce more linking errors than the code being replaced; the concept is explored in [14, 36]. Tools that load the most recent binary compatible version of code were developed for Java [39, 5] and C# [17, 18]. Current JVMs go even further, and support replacing a class by a class of the same signature, as a "fix-and-continue" feature [10].

Dynamic software updating [27] supports type safe dynamic reloading of code whose type may have changed, while the system is running. Proteus [38] 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.

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 [34], 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 DART, European Commission research Directorate IST-01-6-1A. 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.

References

- [1] A. Buckley and S. Drossopoulou. Flexible Dynamic Linking. In *6th Workshop on Formal Techniques for Java-like Programs*, June 2004.
- [2] Martin Abadi, Georges Gonthier, and Benjamin Werner. Choice in Dynamic Linking. In *Proceedings of Foundations of Software Science and Computation Structures*. Springer-Verlag, April 2004.
- [3] Davide Ancona, Giovanni Lagorio, and Elena Zucca. A Formal Framework for Java Separate Compilation. In *ECOOP 2002*, June 2002.
- [4] Davide Ancona and Elena Zucca. A calculus of module systems. *Journal Functional Programming*, 12(3):91–132, 2002.
- [5] Miles Barr and Susan Eisenbach. Safe Upgrading without Restarting. In *IEEE Conference on Software Maintenance ICSM'2003*. IEEE, Sept 2003.
- [6] Gilad Bracha. Packages break strong locality, February 1999. Private Communication, more at <http://www-dse.doc.ic.ac.uk/sue/packages.html>.
- [7] Alex Buckley, Michelle Murray, Susan Eisenbach, and Sophia Drossopoulou. A new linking strategy for .net. Submitted for publication, October 2004.
- [8] Luca Cardelli. Program Fragments, Linking, and Modularization. In *POPL'97 Proceedings*, January 1997.
- [9] Drew Dean. The Security of Static Typing with Dynamic Linking. In *Fourth ACM Conference on Computer and Communication Security*, 1997.
- [10] Mikhail Dimitriev. Hotspot Technology Application for Advanced Profiling. In *ECOOP USE Workshop*, June 2002.
- [11] 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.
- [12] Sophia Drossopoulou and Susan 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.

- [13] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is Java Sound? *Theory and Practice of Object Systems*, 5(1), January 1999.
- [14] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A Fragment Calculus - towards a model of Separate Compilation, Linking and Binary Compatibility. In *LICS Proceedings*, 1999.
- [15] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In *Proceedings of the European Symposium on Programming*. Springer-Verlag, April 2003.
- [16] 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.
- [17] Susan Eisenbach, Vladimir Jurisic, and Chris Sadler. Managing the Evolution of .NET Programs. In *6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems, FMOODS'2003*, volume 2884 of *LNCS*, pages 185–198. Springer-Verlag, Nov. 2003.
- [18] Susan Eisenbach, Dilek Kayhan, and Chris Sadler. Keeping Control of Reuseable Components. In *IEEE Working Conf on Component Deployment*, June 2004.
- [19] G. Fenton and E. Felton. *Securing Java Getting Down to Business with Mobile Code*. John Wiley and Sons, 1999.
- [20] Kathleen Fisher, John Reppy, and Jon Riecke. A Calculus for Compiling and Linking Classes. In *ESOP Proceedings*, March 2000.
- [21] Matthew Flatt, Shiram Krishnamurthi, and Matthias Felleisen. Classes and Mix-ins. In *POPL Proceedings*, January 1998.
- [22] Ira Forman, Michael Conner, Scott Danforth, and Larry Raper. Release-to-Release Binary Compatibility in SOM. In *OOPSLA Proceedings*, October 1995.
- [23] Stephen N. Freund and J. C. Mitchell. A Formal Framework for the Java Bytecode Language and Verifier. In *OOPSLA Proceedings*, November 1999.
- [24] Stephen N. Freund and J. C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. In *OOPSLA Proceedings*, October 1998.
- [25] Andrew Gordon and Don Syme. Typing a multi-language intermediate code. In *Principles of programming Languages 2001*, pages 248–260. ACM Press, 2001.
- [26] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996.
- [27] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic Software Updating. In *Programming Language Design and Implementation*. ACM, 2001.
- [28] Thomas Jensen, Daniel Le Metayer, and Tommy Thorn. A Formalization of Visibility and Dynamic Loading in Java. In *IEEE ICCL*, 1998.
- [29] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the JavaTM Virtual Machine. In *OOPSLA Proceedings*, October 1998.

- [30] Tim Lindholm and Frank Yellin. *The Java Virtual Machine*. Addison-Wesley, 1999.
- [31] Type Safety and Security. In *MSDN Magazine*, 2001.
- [32] M. Pietrek. Avoiding DLL Hell: Introducing Application Metadata in the Microsoft .NET Framework. In *MSDN Magazine*, msdn.microsoft.com/, October 2000.
- [33] S. Pratschner. Simplifying Deployment and Solving DLL Hell with the .NET Framework. msdn.microsoft.com/, November 2001.
- [34] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A Formal Specification of JavaTM Class Loading. In *OOPSLA'2000*, November 2000.
- [35] Vijay Saraswat. Java is not type-safe. Technical report, AT&T Rresearch, 1997. <http://www.research.att.com/~vj/bug.html>.
- [36] David Wragg Sophia Drossopoulou and Susan Eisenbach. What is Java binary compatibility? In *Proc. of OOPSLA*, volume 33, pages 341–358, 1998.
- [37] Raymie Stata and Martin Abadi. A Type System For Java Bytecode Subroutines. In *POPL'98 Proceedings*, January 1998.
- [38] Gareth Stoye, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtii. *Mutatis Mutandis: Safe and flexible dynamic software updating*. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, January 2005.
- [39] Chris Sadler Susan Eisenbach and Shakil Shaikh. Evolution of Distributed Java Programs. In *IEEE Working Conf on Component Deployment*, June 2002.
- [40] Donald Syme. Proving Java Type Sound. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer, 1999.
- [41] T. Van Vleck(ed.). Multics Home. www.multicians.org/, August 2002.
- [42] Joe Wells and Rene Vestergaard. Confluent Equational Reasoning for Linking with First-Class Primitive Modules. In *ESOP Proceedings*, March 2000.
- [43] Elena Zucca, S. Fagorzi, and Davide Ancona. Modeling Multiple Class Loaders by a Calculus for Dynamic Linking. In *Proceedings of the 2004 ACM Symposium on Applied Computing*. ACM Press, March 2004.

A Examples

A.1 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 figure 12.

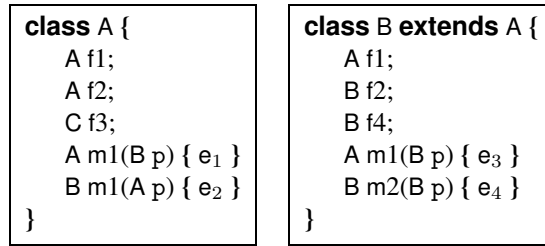


Figure 12: Example program demonstrating layout

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, 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.

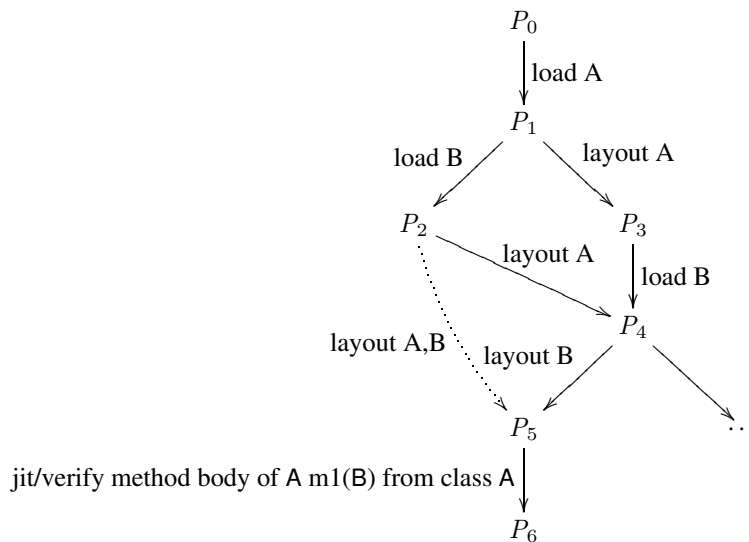


Figure 13 shows a global context W which describes these classes. Also, it shows a possible sequence of programs involved in execution, and the contents of these programs.

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 $W \vdash P_1 \leq P_0$ holds.

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

We then have $W \vdash P_4 \leq P_3$ through loading of B, and $W \vdash P_5 \leq P_4$ 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

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|--|----------------|--|--|----------------|--|--|----------------|--|------------------|----------------|--|--|----------------|--|--|----------------|--|---------------|--|--|--|--|--|---------------|--|--|--|--|--|--|---|------------|-----------------------------------|--|--|-----------------------------------|--|--|-----------------------------------|--|------------|-----------------------------------|--|--|-----------------------------------|--|--|-----------------------------------|--|---------|--------------------------------------|--|--|--------------------------------------|--|---------|--------------------------------------|--|--|--------------------------------------|--|--|--------------------------------------|--|---------|-----------------|--|--|-----------------|--|---------|-----------------|--|--|-----------------|--|--|-----------------|--|
| <p>W</p> <p>$A \mapsto \langle \text{Object}, \bar{\kappa}_1, \bar{\mu}_1 \rangle$ $B \mapsto \langle A, \bar{\kappa}_2, \bar{\mu}_2 \rangle$</p> <hr/> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">$\bar{\kappa}_1$</td> <td style="width: 10%;">$f1 \mapsto A$</td> <td style="width: 80%;"></td> </tr> <tr> <td></td> <td>$f2 \mapsto A$</td> <td></td> </tr> <tr> <td></td> <td>$f3 \mapsto C$</td> <td></td> </tr> <tr> <td>$\bar{\kappa}_2$</td> <td>$f1 \mapsto A$</td> <td></td> </tr> <tr> <td></td> <td>$f2 \mapsto B$</td> <td></td> </tr> <tr> <td></td> <td>$f4 \mapsto B$</td> <td></td> </tr> </table> <hr/> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">$\bar{\mu}_1$</td> <td style="width: 10%;">$\langle m1, A, B \rangle \mapsto e_1$</td> <td style="width: 80%;"></td> </tr> <tr> <td></td> <td>$\langle m1, B, A \rangle \mapsto e_2$</td> <td></td> </tr> <tr> <td>$\bar{\mu}_2$</td> <td>$\langle m1, A, B \rangle \mapsto e_3$</td> <td></td> </tr> <tr> <td></td> <td>$\langle m2, B, B \rangle \mapsto e_4$</td> <td></td> </tr> </table> | $\bar{\kappa}_1$ | $f1 \mapsto A$ | | | $f2 \mapsto A$ | | | $f3 \mapsto C$ | | $\bar{\kappa}_2$ | $f1 \mapsto A$ | | | $f2 \mapsto B$ | | | $f4 \mapsto B$ | | $\bar{\mu}_1$ | $\langle m1, A, B \rangle \mapsto e_1$ | | | $\langle m1, B, A \rangle \mapsto e_2$ | | $\bar{\mu}_2$ | $\langle m1, A, B \rangle \mapsto e_3$ | | | $\langle m2, B, B \rangle \mapsto e_4$ | | <p>P_1 $A \mapsto \langle \text{Object}, \bar{\kappa}_1, \bar{\mu}_1 \rangle$</p> <p>$P_2$ $A \mapsto \langle \text{Object}, \bar{\kappa}_1, \bar{\mu}_1 \rangle$ $B \mapsto \langle A, \bar{\kappa}_2, \bar{\mu}_2 \rangle$</p> <p>$P_3$ $A \mapsto \langle \text{Object}, \kappa_1, \mu_1, \nu_1 \rangle$ $100 \mapsto \langle A, B, e_1 \rangle$ $101 \mapsto \langle B, A, e_2 \rangle$</p> <p>$P_4$ $A \mapsto \langle \text{Object}, \kappa_1, \mu_1, \nu_1 \rangle$ $B \mapsto \langle A, \bar{\kappa}_2, \bar{\mu}_2 \rangle$ $100 \mapsto \langle A, B, e_1 \rangle$ $101 \mapsto \langle B, A, e_2 \rangle$</p> <p>$P_5$ $A \mapsto \langle \text{Object}, \kappa_1, \mu_1, \nu_1 \rangle$ $B \mapsto \langle A, \kappa_2, \mu_2, \nu_2 \rangle$ $100 \mapsto \langle A, B, e_1 \rangle$ $101 \mapsto \langle B, A, e_2 \rangle$ $102 \mapsto \langle A, B, e_3 \rangle$ $103 \mapsto \langle B, B, e_4 \rangle$</p> <p>$P_6$ $A \mapsto \langle \text{Object}, \kappa_1, \mu_1, \nu_1 \rangle$ $B \mapsto \langle A, \kappa_2, \mu_2, \nu_2 \rangle$ $100 \mapsto e'_1$ $101 \mapsto \langle B, A, e_2 \rangle$ $102 \mapsto \langle A, B, e_3 \rangle$ $103 \mapsto \langle B, B, e_4 \rangle$</p> | <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">κ_1</td> <td style="width: 10%;">$\langle f1, A \rangle \mapsto 1$</td> <td style="width: 80%;"></td> </tr> <tr> <td></td> <td>$\langle f2, A \rangle \mapsto 2$</td> <td></td> </tr> <tr> <td></td> <td>$\langle f3, C \rangle \mapsto 3$</td> <td></td> </tr> <tr> <td>κ_2</td> <td>$\langle f1, A \rangle \mapsto 4$</td> <td></td> </tr> <tr> <td></td> <td>$\langle f2, B \rangle \mapsto 5$</td> <td></td> </tr> <tr> <td></td> <td>$\langle f4, B \rangle \mapsto 6$</td> <td></td> </tr> </table> <hr/> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">μ_1</td> <td style="width: 10%;">$\langle m1, A, B \rangle \mapsto 0$</td> <td style="width: 80%;"></td> </tr> <tr> <td></td> <td>$\langle m1, B, A \rangle \mapsto 1$</td> <td></td> </tr> <tr> <td>μ_2</td> <td>$\langle m1, A, B \rangle \mapsto 0$</td> <td></td> </tr> <tr> <td></td> <td>$\langle m1, B, A \rangle \mapsto 1$</td> <td></td> </tr> <tr> <td></td> <td>$\langle m2, B, B \rangle \mapsto 2$</td> <td></td> </tr> </table> <hr/> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">ν_1</td> <td style="width: 10%;">$0 \mapsto 100$</td> <td style="width: 80%;"></td> </tr> <tr> <td></td> <td>$1 \mapsto 101$</td> <td></td> </tr> <tr> <td>ν_2</td> <td>$0 \mapsto 102$</td> <td></td> </tr> <tr> <td></td> <td>$1 \mapsto 101$</td> <td></td> </tr> <tr> <td></td> <td>$2 \mapsto 103$</td> <td></td> </tr> </table> | κ_1 | $\langle f1, A \rangle \mapsto 1$ | | | $\langle f2, A \rangle \mapsto 2$ | | | $\langle f3, C \rangle \mapsto 3$ | | κ_2 | $\langle f1, A \rangle \mapsto 4$ | | | $\langle f2, B \rangle \mapsto 5$ | | | $\langle f4, B \rangle \mapsto 6$ | | μ_1 | $\langle m1, A, B \rangle \mapsto 0$ | | | $\langle m1, B, A \rangle \mapsto 1$ | | μ_2 | $\langle m1, A, B \rangle \mapsto 0$ | | | $\langle m1, B, A \rangle \mapsto 1$ | | | $\langle m2, B, B \rangle \mapsto 2$ | | ν_1 | $0 \mapsto 100$ | | | $1 \mapsto 101$ | | ν_2 | $0 \mapsto 102$ | | | $1 \mapsto 101$ | | | $2 \mapsto 103$ | |
| $\bar{\kappa}_1$ | $f1 \mapsto A$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $f2 \mapsto A$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $f3 \mapsto C$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| $\bar{\kappa}_2$ | $f1 \mapsto A$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $f2 \mapsto B$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $f4 \mapsto B$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| $\bar{\mu}_1$ | $\langle m1, A, B \rangle \mapsto e_1$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $\langle m1, B, A \rangle \mapsto e_2$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| $\bar{\mu}_2$ | $\langle m1, A, B \rangle \mapsto e_3$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $\langle m2, B, B \rangle \mapsto e_4$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| κ_1 | $\langle f1, A \rangle \mapsto 1$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $\langle f2, A \rangle \mapsto 2$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $\langle f3, C \rangle \mapsto 3$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| κ_2 | $\langle f1, A \rangle \mapsto 4$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $\langle f2, B \rangle \mapsto 5$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $\langle f4, B \rangle \mapsto 6$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| μ_1 | $\langle m1, A, B \rangle \mapsto 0$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $\langle m1, B, A \rangle \mapsto 1$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| μ_2 | $\langle m1, A, B \rangle \mapsto 0$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $\langle m1, B, A \rangle \mapsto 1$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $\langle m2, B, B \rangle \mapsto 2$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ν_1 | $0 \mapsto 100$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $1 \mapsto 101$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ν_2 | $0 \mapsto 102$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $1 \mapsto 101$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | $2 \mapsto 103$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Figure 13: Example demonstrating table layout

address 100. Thus, we have that $P_5, e_1 \rightsquigarrow_{W, \text{this} \mapsto A, p \mapsto B} P'', e'_1, t$ and $P'', t, A \rightsquigarrow_W P_6$. So, we also have that $W \vdash P_6 \leq P_5$.