Towards an abstract model of Java dynamic linking and verification

Sophia Drossopoulou
Department of Computing, Imperial College, London

Abstract

We suggest a model for dynamic linking and verification as in Java. We distinguish five components in a Java implementation: evaluation, resolution, loading, verification, and preparation – with their associated checks. We demonstrate how these five together guarantee type soundness.

We take an abstract view, and base our model on a language nearer to Java source than to bytecode. We consider the following features of Java: classes, subclasses, fields and hiding, methods and inheritance, and interfaces.

1 Introduction

Java's recent spectacular success is partly due to its novel approach to code deployment. Rather than compiling and linking a fixed piece of code for a set target machine, Java is compiled to bytecode[18], that can be executed on several platforms, and can link further code on demand: This approach, however, creates opportunities for malicious attacks. The security of Java greatly depends on the safety of the type system [4].

As it is bytecode that is executed rather than source code, and as bytecode is not always the product of compilation, Java security lies primarily with the bytecode verifier, which was formalized as as a type inference system where stack locations have types on a per-instruction basis, [22, 11, 10, 19]. On the other hand, [21] reported security flaws due to inconsistencies between loaders, which were rectified in later releases, as described in [17]. An operational semantics for multiple loaders is given in [14].

Thus, various components of Java and the virtual machine have been studied at considerable depth in isolation, but, except for this paper and [20] their interplay has not yet been formalized.

We attempt a synthesis, and consider the complete process, *i.e.* evaluation, loading, verification, preparation and resolution in a typed setting. We base our model on a language that is very near to Java source, rather than the bytecode, as in [20].

Our model is therefore useful for source language programmers: Even if they do not program in bytecode, and do not download unverified bytecode, they may become aware of these issues, and may trigger verification, resolution and loading errors.[1]

We distinguish the checks performed by verification and resolution, and demonstrate their dependencies: Resolution checks do not guarantee consistency unless applied on verified code, nor are verification checks sufficient unless later supported by resolution checks. Our model clarifies which situation will throw which exceptions, a question that is not unambiguously answered in [12, 18], and it demonstrates how execution of unverified code may corrupt the store.

A clear understanding of these checks and their interplay is crucial for the design of new binary formats for Java. In fact, while most Java implementations use the class format [18], any format satisfying the properties outlined in ch 13.1 of [12] may be used instead.

1.1 Overview of Java verification and dynamic linking, and of our formalization

In traditional programming languages, *e.g.* Ada, Modula-2, the compiler checks all type-related requirements, and produces code which does not contain type information. If the various components of a program have been compiled in an order consistent with their dependencies (dependencies through imports or inheritance) then execution is guaranteed to be sound with respect to types. Thus, before execution, the program is linked eagerly, all external references are resolved and type-checked. Execution has therefore the form

$$\mathsf{e}, \sigma, \mathsf{Code} \rightsquigarrow \mathsf{e}', \sigma', \mathsf{Code}$$

i.e. it takes place in the context of fixed Code, and modified the expression e and the store σ.

Also, if the expression and state are well-formed in the context of Code, and e is not ground, then execution will continue with a well-formed expression, unless a program exception is thrown. We call program exceptions those

[1] By compiling modified Java classes without recompiling all importing classes one may obtain bytecode that does not verify. Also, execution sometimes does not attempt to verify local classes.

term	meaning	definition
$\mathcal{L}$	language of loaded code	fig 3
$\mathcal{P}$	language of prepared code	fig 3
e	a term (identical in $\mathcal{L}$ and $\mathcal{P}$)	
σ	a store, mapping identifiers and integers to identifiers or integers	sect. 5.1
L	loaded code, from $\mathcal{L}$	
P	prepared code from $\mathcal{P}$	
$\mathsf{e}, \sigma, \mathsf{P}, \mathsf{L} \rightsquigarrow \mathsf{e}', \sigma', \mathsf{PP}', \mathsf{L}'$	e, σ rewrite to e', σ', prepared code augmented by P', loaded code becomes L'	fig 4
$\sqsubset \cdot \sqsupset^{exp}$	expression context, propagates to sub-expression	fig 5
$\sqsubset \cdot \sqsupset^{nll}$	null context, may throw exception	fig 5
$\sqsubset \cdot \sqsupset^{typ}$	type context, may cause loading and verification	fig 5
$\mathsf{P}, \mathsf{L} \vdash \mathsf{c} \leq_{clss} \mathsf{c}'$	c is a subclass of c' in context of P, L	fig 6
$\mathsf{P}, \mathsf{L} \vdash \mathsf{c} \leq_{impl} \mathsf{i}$	c implements i in context of P, L	fig 6
$\mathsf{P}, \mathsf{L} \vdash \mathsf{i} \leq_{intf} \mathsf{i}'$	i is a subinterface of i' in context of P, L	fig 6
$\vdash \mathsf{P}, \mathsf{L} \diamond_a$	the subclass/subinterface relationship in P, L is acyclic	fig 6
$\vdash \mathsf{P}, \mathsf{L} \diamond_{sups}$	P, L contain all superclasses/superinterfaces of classes/interfaces defined in PL	fig 6
$\mathsf{P}, \mathsf{L} \vdash_{v} \mathsf{L}' \diamond \underset{loads}{\leftharpoondown} \mathsf{L}''$	verifier checks that L' is well formed in context of P, L, while loading L''	fig 7
$\mathsf{P}, \mathsf{L} \vdash_{v} \mathsf{t} \leq \mathsf{t}' \underset{loads}{\leftharpoondown} \mathsf{L}'$	verifier checks that t widens to t' in context of P, L, while loading L'	fig 7
$\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{v} \mathsf{e} : \mathsf{t} \underset{loads}{\leftharpoondown} \mathsf{L}'$	verifier checks that e has type t in context of P, L, while loading L'	fig 7
E	environment for the declaration of variables	fig 8
$\mathsf{P}, \mathsf{L} \vdash \mathsf{t} \leq \mathsf{t}'$	t widens to t' in the context of the prepared code P, and loaded code L	fig 9
$\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash \mathsf{e} : \mathsf{t}$	e has type t in the context of the prepared code P, and environment E	fig 9
$\mathsf{P}, \mathsf{L} \vdash \mathsf{P}' \diamond$	P' is well-formed in the context of P and L	fig 9
$\mathsf{L} \vdash \mathsf{P} \diamond$	P is well-formed in the context of L	fig 9
$\sigma, \mathsf{P} \vdash_{cw} \gamma : \mathsf{t}$	value γ conforms weekly to type t in context of P	fig 10
$\sigma, \mathsf{P} \vdash_{c} \alpha \diamond$	the object stored at α in σ is well-formed (conforms strongly)	fig 10
$\mathsf{P}, \mathsf{E} \vdash_{c} \sigma \diamond$	all objects in σ are well-formed, and agree to their declarations in E	fig 10
$\mathsf{P}, \mathsf{L} \vdash \mathsf{t} \leq \mathsf{t}'$	t widens to t' in the context of P and L	fig 10
$\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{e} : \mathsf{t}$	run-time expression e has type t in store σ in the context of P, L, E	fig 10
$ld(\mathsf{t}, \mathsf{P}, \mathsf{L})$	loading	fig 11
$pr(\mathsf{L}, \mathsf{P})$	preparation	fig 11
$\mathcal{F}o(\mathsf{f}, \mathsf{c}, \mathsf{t}, \mathsf{P})$	the offset of field f with type t in class c	fig 12
$\mathcal{F}s(\mathsf{c}, \mathsf{P})$	al fields with types and offsets, defined or inherited in class c	fig 12
$\mathcal{M}o(\mathsf{m}, \mathsf{c}, \mathsf{t}_2, \mathsf{t}_1, \mathsf{P})$	the offset of method m with argument type t_2 and result type t_1 in class c	fig 12
$\mathcal{M}e(\beta, \mathsf{c}, \mathsf{P})$	the method body at offset β in class c	fig 12
$\mathcal{M}o^{\mathsf{i}}(\mathsf{m}, \mathsf{i}, \mathsf{t}_2, \mathsf{t}_1, \mathsf{P})$	the offset of method m with argument type t_2 and result type t_1 in interface i	fig 12
$\vdash \mathsf{L} \diamond_u$, $\vdash \mathsf{P} \diamond_u$, $\vdash \mathsf{P}, \mathsf{L} \diamond_u$	definitions in L, or in P, or in L and P are unambiguous	omitted
$\mathsf{L(t)}$, $\mathsf{P(t)}$, $\mathsf{PL(t)}$	look-up class or interface t in L, or in P, or in L and P	omitted

Figure 1. Concepts defined in this paper

caused by the logic of the program, eg division by zero, null pointer dereferencing *etc*.

Java on the other hand, does not require the complete program to have been linked before execution. During execution it is possible that a class is needed, which is part of the current code. If bytecode for the class name can be found, and verified, then the code is enriched with the new class, otherwise, a load-exception or a verification-exception is thrown.

The Java approach is even lazier, in the sense that Code consists of a verified part P, and a loaded part L, which was loaded in order to support verification of P. We consider language $\mathcal{L}$, which stands for *loaded* binary programs, and $\mathcal{P}$, which stands for verified and *prepared* binary programs, *c.f.* section 2.

Verification checks that the subtype relations required in some code are satisfied, but does *not check* the presence of fields or methods referred to in some piece of code. That is checked only when and if the method or field are accessed; if they cannot be found, then a resolution-exception is thrown.

Therefore, we describe execution in terms of expressions e, states σ, verified code P, and loaded but not verified code L. It has the general form

$$\mathsf{e}, \sigma, \mathsf{P}, \mathsf{L} \rightsquigarrow \mathsf{e}', \sigma', \mathsf{PP}', \mathsf{L}'$$

thus describing that the expression may be rewritten, the state may be modified, code may be loaded, and some of the loaded code may be verified and prepared. The possible errors are program exceptions, loading exceptions, verification exceptions, and resolution exceptions.

We classify execution into the following five components:

- *evaluation* corresponds to execution as in most programming languages,
- *resolution* describes the process of resolving references to fields and methods,
- *loading* is the process of loading class descriptions necessary for the verification of further classes,
- *verification* is the process of verifying $\mathcal{L}$ code
- *preparation* turns verified $\mathcal{L}$ code into $\mathcal{P}$ code.

We demonstrate these components in terms of an example. Consider the following high level view of byte code method call: **new** A[A, **int**, **void**].m(3)
which stands for the call of a method m, defined in class A, which takes an **int** parameter, and returns **void**,[2], and where the receiver is a new object of class A.

[2]Method calls in Java bytecode contain the signature of the method.

Assume that class A is not defined in L, nor in P. If **class** A can not be found, then a load error is thrown, otherwise A is *loaded*, and L is extended. Then class A is *verified*, which means that all its method bodies, and all its superclasses will be checked, and all required subtype relationships will be checked. Assume that class A had a method

void m(**int** x){ B aB; aB = **new** C; aB[B, **int**].f = x }

The term aB[B, **int**].f indicates selection from aB of a field f defined in class B with type **int**.

Verification of the above method body requires that class C is a subtype of B. Assume that C has not been loaded nor verified yet. Then it will get loaded together with all its superclasses. If those include B, then verification will be successful. This is and example of a class that is loaded but *not verified*.

We represent verification through a judgement

$$\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{v} \mathsf{e} \ : \ \mathsf{t} \quad \underset{loads}{\leftharpoondown} \mathsf{L}',$$

which means that the expression e could be verified in the context of prepared P, loaded L, and environment E, and required further binaries L′ to be loaded.

Now consider execution of the method body for m. The creation of the object of class C requires *preparation* of the class C. Preparation determines the layout of the objects of that class and the layout of the method look-up table of that class, ensuring that the offsets for inherited fields and methods coincide with those of the superclasses.

When the assignment aB[B, **int**].f = x is executed, the filed access aB[B, **int**].f is *resolved*. If class B does not have a field f of type **int**, then a resolution exception is thrown. Otherwise, resolution returns the offset of **int** f from class B. This offset is used to access the field in aB, which happens to belong to class C. But because C is a subclass of B, and has been prepared, it will have the inherited field at the same offset as B, and so the assignment will not break the consistency of the object.

If however, the method body had not been verified, and C was not a subclass of B, or if resolution could be fooled, then the integrity of the object could be violated. Thus, the above example demonstrates how the verification and resolution checks complement each other.

We represent consistency of states with prepared code through $\mathsf{P}, \mathsf{E} \vdash_c \sigma \diamond$ and types for run-time expressions through $\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_r \sigma, \mathsf{e} \ : \ \mathsf{t}$. In section 4 we prove a subject reduction and progress lemma, which guarantees for well-formed P, state σ consistent with P and L, and well-typed e that execution will either produce a well-typed e′, or a null pointer exception (if a null pointer is de-referenced), a loader exception (if requested classes could not be found, or were circular), or a verifier exception (if verification of requested classes unsuccessful), or a member absent exception (if a non-existing method or field was accessed). In all cases it will preserve the consistency of σ' which is crucial for safety. Furthermore, execution will never get stuck.

In figure 1 we list all judgements and functions defined in the paper, with a brief description of their intention, and the place of their definition.

The treatment of interfaces

In order to establish that required subtype relationships are satisfied, verification looks up the appropriate classes.

However, if the required subtype relationships involve interfaces, then these relationships are automatically assumed to hold and are *not* checked!

> Apparently overawed by the multiplicity of parents possible in a Java interface hierarchy, the implementors of Sun's verifier ... abdicated responsibility for type checking involving the use of interfaces. Instead, ..., the burden of checking for compatibility, ... passed implicitly to the run-time system
> Philipp Yelland [25]

Thus, at run-time these subtype requirements need to be checked, and execution of interface method calls will check the satisfaction of the associated subtype relationship. Again, we see that checks from two different JVM components complement each other, and in slightly different ways for classes than for interfaces.

2 The languages $\mathcal{L}$ and $\mathcal{P}$

The binary language $\mathcal{L}$ presents an abstract view of the Java bytecode. In order to keep the discussion simple, we only consider classes, subclasses, interfaces, subinterfaces, assignment, method overloading and inheritance, field inheritance and hiding.[3] Even though our examples use sequential statements, we have not included them in the $\mathcal{L}$- and $\mathcal{P}$-syntax, as they can be easily encoded by extra methods. In order to simplify the presentation, all methods have one argument, called x.

The only types we consider are classes, interfaces and int; these demonstrate several interesting properties of the Java system. Interfaces introduce multiple subtyping. More interestingly, subtyping introduced through interfaces is dealt with differently from subtyping introduced through subclassing: as we shall see, the verifier assumes an interface to be a supertype of *any* type, whereas it considers a class to be a supertype of its loaded subclasses only; therefore, at runtime subclasses are not checked for instance

[3] $\mathcal{L}$ is a similar language to language Javacito[16] or the Java subset from [8]; it is larger than [13] because it considers imperative features, overloading and interfaces; and, though at a different abstraction level than [20], it is larger because it studies interfaces .

p	$::=$	def^*
def	$::=$	**interface** i **ext** i^* { $methHd^*$ }
		class c **ext** c' **impl** i^* { $field^*$ $meth^*$ }
$methHd$	$::=$	$type\ m(type$ x $)$
$meth$	$::=$	$type\ m(type$ x $)\{exp\}$ δ
$field$	$::=$	$type\ f$ δ
exp	$::=$	exp **[**$type$,$type$,$type$**]**.m (exp)
	|	exp **[**$type$,$type$,$type$**]**$^{\mathbf{i}}$.m (exp)
	|	exp **[**$type$,$type$**]**.f
	|	var = exp
	|	**new** c
	|	**this**
	|	var
	|	γ
var	$::=$	x | z | exp **[**$type$,$type$**]**.f
$type$	$::=$	c | int | i
δ	$::=$	ϵ in $\mathcal{L}$
δ	$::=$	β in $\mathcal{P}$
β	$::=$	**1** | **2** | ...
α	$::=$	**0** | β
γ	$::=$	α | **-1** | **-2** | ...
c, i, m, f, z	$::=$	Ident

and where
c are class names, i are interface names
m are method names, f are field names,
α are addresses, β are offsets, γ are integer values.

Figure 3. The syntax of $\mathcal{L}$ and of $\mathcal{P}$

method calls, but subtypes are checked for interface method calls. Also, the type **int** and the address calculations during execution open the possibility of pitfalls, which, as we shall demonstrate, are averted by verification and the resolution checks.

In $\mathcal{L}$, as in the bytecode, field accesses and method calls are annotated by descriptors. Field access[4] has the form $e_1[t_1,t_2].f$, where t_1 is the class containing the field definition, and t_2 the type of that field. Instance method calls[5] have the form $e_1[t_1,t_2,t_3].m(e_2)$, where t_1 is the class containing the method definition, t_2 is the type of the method's argument, and t_3 is the result type. Similarly, interface method calls[6] have the form $e[t_1,t_2,t_3]^{\mathbf{i}}.m(e_2)$; where t_1 is the interface containing the method definition, t_2 is the argument type, and t_3 is the result type.

Values are either integers or addresses of objects. Addresses are represented by positive integers and are denoted by α or α'; the null pointer **null** is denoted by **0**. Integer values, whether they stand for addresses or for integers, are

[4] corresponding to the bytecode instructions getfield and putfield
[5] corresponding to the bytecode instruction invokevirtual
[6] corresponding to the bytecode instruction invokeinterface

```
LPh =
class Phil ext Object impl ϵ {
    int age
    Phil like
    Book think( FrPhil x)
        { ... x[FrPhil,Food].like = new Pear ... }
    Phil eat( Food x){ ... x = new Food ... }
}

LFrPh =
class FrPhil ext Phil impl ϵ {
    Food like


    Phil eat( Food x){ ... x = new Pear ... }
    Food think( FrPhil x)
        { new Pear[Food, Salt, Food].cook( new Salt) }
}
```

```
PPh =
class Phil ext Object impl ϵ {
    int age                                              1
    Phil like                                            2
    Book think( FrPhil x)
        { ... x[FrPhil,Food].like = new Pear ... }       1
    Phil eat( Food x){ ... x = new Food ... }            2
}

PFrPh =
class FrPhil ext Phil impl ϵ {
    Food like                                            3
    Book think( FrPhil x)
        { ... x[FrPhil,Food].like = new Pear ... }       1
    Phil eat( Food x){ ... x = new Pear ... }            2
    Food think( FrPhil x)
        { new Pear[Food, Salt, Food].cook( new Salt) }   3
}
```

Figure 2. An example in $\mathcal{L}$, and the corresponding example in $\mathcal{P}$

denoted by γ, γ' *etc.*

Figure 3 contains the syntax of $\mathcal{L}$ and of $\mathcal{P}$; figure 2 contains an example in $\mathcal{L}$ and the corresponding example in $\mathcal{P}$. The example is a variation of the one given in [5]: Phil-osophers have an age, they like other Phil-osophers, and produce Book-s when they think; whereas FrPhil-osophers like Food, and produce Food when they think.

Notice, that the field like in FrPhil "shadows" that of class Phil. Objects of class FrPhil contain three fields, *i.e.* age and like from class Phil, and like from class FrPhil.[7] Field selection is determined by the type annotations. For example, x[Phil,Phil].like selects the field of type Phil defined in class Phil, whereas x[FrPhil,Food].like selects the field of type Food defined in class FrPhil.

The instance method call x[Phil,FrPhil,Book].think(...) selects from the class Phil the method which takes a FrPhil parameter and returns a Book, whereas x[FrPhil,FrPhil,FrPhil].think(...) selects from the class FrPhil the method which takes a FrPhil parameter and returns a FrPhil.

Contrary to Java source language rules [12], $\mathcal{L}$- and $\mathcal{P}$-methods may have the same identifier and argument type but *different* result type as a method from a superclass, *e.g.* method Book think(FrPhil x) { ... } in class Phil, and method Food think(FrPhil x) { ... } in class FrPhil.[8]

The language $\mathcal{P}$ describes code after preparation; the programs are extended by offset information. Thus, the syntax of expressions in $\mathcal{P}$ is identical to that of expressions in $\mathcal{L}$, except that field declarations are augmented by offsets, determining the field's position in actual objects on the heap, and method definitions are augmented by their offsets, describing the method's position in the method look up tables. Offsets are positive integers, and denoted by β, β' *etc.*

The classes $\mathsf{P_{Ph}}$ and $\mathsf{P_{FrPh}}$ are possible results of the preparation of $\mathsf{L_{Ph}}$, $\mathsf{L_{FrPh}}$: The fields in the subclass (here like in FrPhil) are given distinct offsets to those of the fields in the superclass. All inherited methods (here method Book think(FrPhil x){... } inherited in FrPhil from Phil) appear in the subclass with the same offset, whereas new methods (here method Food think(FrPhil x){...} in FrPhil) are given fresh offsets. Finally, any methods overriding methods from a superclass obtain the overridden method's offset (here method Phil eat(Food x){ ... x = **new** Pear ... } from class FrPhil overrides method Phil eat(Food x){ ... x = **new** Food ... }, and therefore has offset 2).

A basic requirement for $\mathcal{L}$ and $\mathcal{P}$ code is that it should be unambiguous. That is, each class or interface should have at most one definition, in L, or in P, or in L and P together. This is expressed by the judgments $\vdash \mathsf{L} \diamond_u$, or $\vdash \mathsf{P} \diamond_u$ or $\vdash \mathsf{P}, \mathsf{L} \diamond_u$. If these judgments are satisfied, the lookup functions L(c) , or P(c) , or PL(c) , will return the appropriate class or interface body, or ϵ if none is there.[9]

Also, the subclass and subinterface relationship in P and L should acyclic, as expressed by the judgment $\vdash \mathsf{P}, \mathsf{L} \diamond_a$,

[7] It is not required that the field in the subclass has a different type than that in the superclass; for example, it would be legal if like in class FrPhil had type Phil.

[8] Such binaries may be created, *e.g.* through compilation of a class and its subclass, subsequent addition of a method in the superclass, and re-compilation of the superclass without re-compilation of the subclass.

[9] We do not define these judgments and look-up functions since they are standard.

Evaluation

$$\textsc{Propagate}\quad \frac{e,\sigma,P,L \rightsquigarrow e',\sigma',P',L'}{\sqsubset e\sqsupset^{exp},\sigma,P,L \rightsquigarrow \sqsubset e'\sqsupset^{exp},\sigma,P,L}$$

$$\textsc{NullPointErr}\quad \frac{}{\sqsubset \mathbf{0}\sqsupset^{nll},\sigma,P,L \rightsquigarrow \mathsf{NllPErr},\sigma,P,L}$$

$$\textsc{Acc}\quad \frac{z \text{ a variable}}{z,\sigma,P,L \rightsquigarrow \sigma(z),\sigma,P,L}$$

$$\textsc{New}\quad \frac{\begin{array}{l} P(c)\neq\epsilon \\ \alpha \text{ new in } \sigma \\ \mathcal{F}s(c,P)=\{t_1\ f_1\ \beta_1,\ldots t_n\ f_n\ \beta_n\} \\ \sigma'=\sigma[\alpha\mapsto c,\alpha+\beta_1\mapsto\mathbf{0},\ \ldots\ \alpha+\beta_n\mapsto\mathbf{0}]\end{array}}{\mathbf{new}\ c,\sigma,P,L \rightsquigarrow \alpha,\sigma',P,L}$$

$$\textsc{VarAss}\quad \frac{}{z=\gamma,\sigma,P,L \rightsquigarrow \gamma,\sigma[z\mapsto\gamma],P,L}$$

Resolution

$$\textsc{FldAcc1}\quad \frac{\mathcal{F}o(f,t_1,t_2,P)=\beta}{\begin{array}{l}\alpha[t_1,t_2].f,\sigma,P,L \rightsquigarrow \sigma(\alpha+\beta),\sigma,P,L \\ \alpha[t_1,t_2].f=\gamma,\sigma,P,L \rightsquigarrow \gamma,\sigma[\alpha+\beta\mapsto\gamma],P,L\end{array}}$$

$$\textsc{FldAcc2}\quad \frac{\mathcal{F}o(f,t_1,t_2,P)=-\mathbf{2}}{\begin{array}{l}\alpha[t_1,t_2].f,\sigma,P,L \rightsquigarrow \mathsf{ClssChngErr},\sigma,P,L \\ \alpha[t_1,t_2].f=\gamma,\sigma,P,L \rightsquigarrow \mathsf{ClssChngErr},\sigma,P,L\end{array}}$$

$$\textsc{FldAcc3}\quad \frac{\mathcal{F}o(f,t_1,t_2,P)=-\mathbf{1}}{\begin{array}{l}\alpha[t_1,t_2].f,\sigma,P,L \rightsquigarrow \mathsf{NoFldErr},\sigma,P,L \\ \alpha[t_1,t_2].f=\gamma,\sigma,P,L \rightsquigarrow \mathsf{NoFldErr},\sigma,P,L\end{array}}$$

$$\textsc{MethCall1}\quad \frac{\mathcal{M}o(m,t_1,t_2,t_3,P)=-\mathbf{2}}{\alpha[t_1,t_2,t_3].m(\gamma),\sigma,P,L \rightsquigarrow \mathsf{ClssChngErr},\sigma,P,L}$$

$$\textsc{MethCall2}\quad \frac{\mathcal{M}o(m,t_1,t_2,t_3,P)=-\mathbf{1}}{\alpha[t_1,t_2,t_3].m(\gamma),\sigma,P,L \rightsquigarrow \mathsf{NoMethErr},\sigma,P,L}$$

$$\textsc{MethCall3}\quad \frac{\begin{array}{l}\mathcal{M}o(m,t_1,t_2,t_3,P)=\beta \\ \mathcal{M}e(\beta,\sigma(\alpha),P)=e \\ y_1,y_2 \text{ are fresh variables in } \sigma \\ e'=e[x/y_1,\mathbf{this}/y_2] \\ \sigma'=\sigma[y_1\mapsto\gamma,y_2\mapsto\alpha]\end{array}}{\alpha[t_1,t_2,t_3].m(\gamma),\sigma,P,L \rightsquigarrow e',\sigma',P,L}$$

$$\textsc{IntfMethCall1}\quad \frac{\mathcal{M}o^{\mathbf{i}}(m,t_1,t_2,t_3,P)=-\mathbf{2}}{\alpha[t_1,t_2,t_3]^{\mathbf{i}}.m(\gamma),\sigma,P,L \rightsquigarrow \mathsf{ClssChngErr},\sigma,P,L}$$

$$\textsc{IntfMethCall2}\quad \frac{\mathcal{M}o^{\mathbf{i}}(m,t_1,t_2,t_3,P)=-\mathbf{1}}{\alpha[t_1,t_2,t_3]^{\mathbf{i}}.m(\gamma),\sigma,P,L \rightsquigarrow \mathsf{NoMethErr},\sigma,P,L}$$

$$\textsc{IntfMethCall3}\quad \frac{P,L\not\vdash\sigma(\alpha)\ \leq_{impl}\ t_1}{\alpha[t_1,t_2,t_3]^{\mathbf{i}}.m(\gamma),\sigma,P,L \rightsquigarrow \mathsf{ClssChngErr},\sigma,P,L}$$

$$\textsc{IntfMethCall4}\quad \frac{\begin{array}{l}P,L\vdash\sigma(\alpha)\ \leq_{impl}\ t_1 \\ \mathcal{M}o^{\mathbf{i}}(m,t_1,t_2,t_3,P)=\mathbf{0} \\ \mathcal{M}o(m,\sigma(\alpha),t_2,t_3,P)=\beta \\ \mathcal{M}e(\beta,\sigma(\alpha),P)=e \\ y_1,y_2 \text{ are fresh variables in } \sigma \\ e'=e[x/y_1,\mathbf{this}/y_2] \\ \sigma'=\sigma[y_1\mapsto\gamma,y_2\mapsto\alpha]\end{array}}{\alpha[t_1,t_2,t_3]^{\mathbf{i}}.m(\gamma),\sigma,P,L \rightsquigarrow e',\sigma',P,L}$$

Loading

$$\textsc{LoadErr}\quad \frac{\begin{array}{l}P(t)=L(t)=\epsilon \\ ld(t,P,L)=\epsilon,\quad \text{for a loader } ld\end{array}}{\sqsubset t\sqsupset^{typ},\sigma,P,L \rightsquigarrow \mathsf{LoadErr},\sigma,P,L}$$

$$\textsc{Load}\quad \frac{\begin{array}{l}e=\sqsubset t\sqsupset^{typ} \\ P(t)=L(t)=\epsilon \\ ld(t,P,L)=L',\quad \text{for a loader } ld\end{array}}{e,\sigma,P,L \rightsquigarrow e,\sigma,P,LL'}$$

Verification

$$\textsc{VerifErr}\quad \frac{\begin{array}{l}P(t)=\epsilon \\ L_1(t)\neq\epsilon,\ \text{ and } \vdash\ P,L_1\diamond_{sups} \\ \forall L':\ \ P,L\not\vdash_v L_1\diamond\ \ \underset{loads}{\leftharpoondown}\ L'\end{array}}{\sqsubset t\sqsupset^{typ},\sigma,P,L_1L_2 \rightsquigarrow \mathsf{VerifErr},\sigma,P,L_1L_2}$$

Preparation

$$\textsc{VerifAndPrep}\quad \frac{\begin{array}{l}e=\sqsubset t\sqsupset^{typ} \\ P(t)=\epsilon,\ \text{ and } L_1(t)\neq\epsilon,\ \text{ and } \vdash\ P,L_1\diamond_{sups} \\ P,L\vdash_v L_1\diamond\ \ \underset{loads}{\leftharpoondown}\ L' \\ P_1=pr(P,L_1),\quad \text{for a preparation } pr\end{array}}{e,\sigma,P,L_1L_2 \rightsquigarrow e,\sigma,PP_1,L_2L'}$$

Figure 4. Execution

$$
\begin{array}{lll}
\sqsubset\cdot\sqsupset^{exp} & ::= & \sqsubset\cdot\sqsupset[type,type,type]\textbf{.}m\ (exp) \\
 & | & \alpha\ [type,type,type]\textbf{.}m\ (\sqsubset\cdot\sqsupset) \\
 & | & \sqsubset\cdot\sqsupset[type,type,type]^{\mathbf{i}}\textbf{.}m\ (exp) \\
 & | & \alpha\ [type,type,type]^{\mathbf{i}}\textbf{.}m\ (\sqsubset\cdot\sqsupset) \\
 & | & \sqsubset\cdot\sqsupset[type,type]\textbf{.}f \\
 & | & \sqsubset\cdot\sqsupset\ \textbf{=}\ exp \\
 & & \quad \text{if } \sqsubset\cdot\sqsupset \text{ is a non-l-ground variable} \\
 & | & var\ \textbf{=}\ \sqsubset\cdot\sqsupset \\
 & & \quad \text{if } var \text{ is an l-ground variable} \\
\sqsubset\cdot\sqsupset^{nll} & ::= & \sqsubset\cdot\sqsupset[type,type,type]\textbf{.}m\ (exp) \\
 & | & \sqsubset\cdot\sqsupset[type,type,type]^{\mathbf{i}}\textbf{.}m\ (exp) \\
 & | & \sqsubset\cdot\sqsupset[type,type]\textbf{.}f \\
 & | & \sqsubset\cdot\sqsupset[type,type]\textbf{.}f\ \textbf{=}\ \gamma \\
\sqsubset\cdot\sqsupset^{typ} & ::= & \alpha\ [\sqsubset\cdot\sqsupset,type,type]\textbf{.}m\ (\gamma) \\
 & | & \alpha\ [\sqsubset\cdot\sqsupset,type,type]^{\mathbf{i}}\textbf{.}m\ (\gamma) \\
 & | & \alpha\ [\sqsubset\cdot\sqsupset,type]\textbf{.}f \\
 & | & \textbf{new}\ \sqsubset\cdot\sqsupset
\end{array}
$$

Figure 5. Contexts

defined in figure 6.

Last, we call an expression *ground*, if it is a value γ[10], and *l-ground*, if it is an identifier, or has the form $\alpha[t_1,t_2]$.f.

3 Execution

Execution, described in figure 4, is defined in terms of a rewriting relationship on *configurations*, consisting of expression e, store σ, prepared code P, and loaded binary L. The expression and store may be modified, more code may be linked, and further binaries may be loaded. Thus, execution has the form $\mathsf{e},\sigma,\mathsf{P},\mathsf{L} \leadsto \mathsf{e}',\sigma',\mathsf{PP}',\mathsf{L}'$.

In order to give a more concise description of the rewrite semantics, and also, in order to distinguish between routine rewrite rules, and those particular to Java implementation, in figure 5 we introduce three kinds of contexts. Expression contexts, $\sqsubset\cdot\sqsupset^{exp}$, are filled with a sub-expression; their execution propagates execution to this sub-expression, as in rule PROPAGATE. Null-contexts, $\sqsubset\cdot\sqsupset^{nll}$, when filled with **0**, raise an exception when executed as in rule NULLPOINTERERR. Type contexts, $\sqsubset\cdot\sqsupset^{typ}$, are filled with a type name; their execution causes the type to be loaded and prepared if the type is not part of the loaded or the prepared code, as in rules LOAD, LOADERR, VERIF, VERIFERR and VERIFANDPREP.

3.1 The run-time model

States represent stacks and heaps, and contain values for identifiers and addresses – the former model formal parameters[11] and addresses. Addresses point to objects. An object consists of its class (an identifier) and values for its fields. These are either values of type **int** or addresses; both are represented by integers. The symbol $*$ means undefined. Stores thus have the form:

$\sigma : [\ \mathrm{Idnt} \rightarrow (\ \mathrm{int} \cup \{*\}\ )] \cup [\ \mathrm{int} \rightarrow (\ \mathrm{int} \cup \mathrm{Idnt} \cup \{*\}\ )]$.

For a variable z, and address α, the store lookup $\sigma(\mathsf{z})$ describes the value of variable z in σ, whereas $\sigma(\alpha) = \mathsf{c}$ determines that α points to an object of class c. The fields of the object pointed at by α are stored at some offset from α. We call an address α *new* in σ iff $\sigma(\alpha+\gamma) = *, \ \forall\gamma \geq 0$.

Our model of the store is therefore at a lower level than those found in studies of the verifier [22, 10, 20], where objects are indivisible entities, and where there are no address calculations. This allows us to describe the potential damage when executing unverified code; as shown in example in section 5.3, field assignment in unverified code could overwrite any part of the memory.

In the example below, the store σ_0 maps identifier aPh to an object of class Phil, whose field like points to an object of class FrPhil:

σ_0(aPh)	=	5	
σ_0(5)	=	Phil	object of class Phil
σ_0(6)	=	45	field **int** age from Phil
σ_0(7)	=	8	field Phil like from Phil
σ_0(8)	=	FrPhil	object of class FrPhil
σ_0(9)	=	55	field **int** age from Phil
σ_0(10)	=	5	field Phil like from Phil
σ_0(11)	=	**0**	field Food like in FrPhil
σ_0(y)	=	$*$	for y $\notin$ {aPh, 5...11}

3.2 An example

The following expression e_1 represents the body of the method Food think(FrPhil x) from class FrPhil, *i.e.*

$\mathsf{e}_1 \equiv$ **new** Pear[Food, Salt, Food]**.**cook(**new** Salt)

The expression e_1 is "well-behaved"if the following requirements are satisfied:

R1 class Pear exists,

R2 Pear is a subclass of Food,

R3 class Salt exists,

R4 Salt is a subtype of Salt,

R5 class Pear has a method Food cook(Salt x){... },

R6 the method Food cook(Salt x){... } from class Pear is "well-behaved" and returns an object of a subtype of Food.

[10]and thus also if it is an address α

[11]In Java, assignment to formal parameters does not overwrite the actual parameter

$$\frac{\vdash \mathsf{P},\mathsf{L}\ \diamond_u \qquad \mathsf{PL}(\mathsf{c}) = \textbf{class}\ \mathsf{c}\ \textbf{ext}\ \mathsf{c}'\ \textbf{impl}\ \ldots \mathsf{i} \ldots \{\ldots\}}{\mathsf{P},\mathsf{L} \vdash \mathsf{c}\ \leq_{clss}\ \mathsf{c} \qquad \mathsf{P},\mathsf{L} \vdash \mathsf{c}\ \leq_{clss}\ \mathsf{c}' \qquad \mathsf{P},\mathsf{L} \vdash \mathsf{c}\ \leq_{impl}\ \mathsf{i}}$$

$$\frac{\vdash \mathsf{P},\mathsf{L}\ \diamond_u \qquad \mathsf{PL}(\mathsf{i}) = \textbf{interface}\ \mathsf{i}\ \textbf{ext}\ \ldots \mathsf{i}' \ldots \{\ldots\}}{\mathsf{P},\mathsf{L} \vdash \mathsf{i}\ \leq_{intf}\ \mathsf{i} \qquad \mathsf{P},\mathsf{L} \vdash \mathsf{i}\ \leq_{intf}\ \mathsf{i}'}$$

$$\frac{\mathsf{P},\mathsf{L} \vdash \mathsf{c}\ \leq_{clss}\ \mathsf{c}'' \qquad \mathsf{P},\mathsf{L} \vdash \mathsf{c}''\ \leq_{clss}\ \mathsf{c}'}{\mathsf{P},\mathsf{L} \vdash \mathsf{c}\ \leq_{clss}\ \mathsf{c}'}$$

$$\frac{\mathsf{P},\mathsf{L} \vdash \mathsf{i}\ \leq_{intf}\ \mathsf{i}' \qquad \mathsf{P},\mathsf{L} \vdash \mathsf{c}\ \leq_{impl}\ \mathsf{i} \qquad \mathsf{P},\mathsf{L} \vdash \mathsf{c}'\ \leq_{clss}\ \mathsf{c}}{\mathsf{P},\mathsf{L} \vdash \mathsf{c}'\ \leq_{impl}\ \mathsf{i}'}$$

$$\frac{\begin{array}{l}\mathsf{P},\mathsf{L} \vdash \mathsf{c}\ \leq_{clss}\ \mathsf{c}'\ \text{and}\ \mathsf{P},\mathsf{L} \vdash \mathsf{c}'\ \leq_{clss}\ \mathsf{c} \quad \Rightarrow \quad \mathsf{c} = \mathsf{c}' \\ \mathsf{P},\mathsf{L} \vdash \mathsf{i}\ \leq_{intf}\ \mathsf{i}'\ \text{and}\ \mathsf{P},\mathsf{L} \vdash \mathsf{i}'\ \leq_{intf}\ \mathsf{i} \quad \Rightarrow \quad \mathsf{i} = \mathsf{i}'\end{array}}{\vdash \mathsf{P},\mathsf{L}\ \diamond_a}$$

$$\frac{\begin{array}{l}\mathsf{P},\mathsf{L} \vdash \mathsf{c}\ \leq_{clss}\ \mathsf{c}' \quad \Rightarrow \quad \mathsf{c}' = \mathsf{Object},\ \text{or}\ \mathsf{PL}(\mathsf{c}') \neq \epsilon \\ \mathsf{P},\mathsf{L} \vdash \mathsf{i}\ \leq_{intf}\ \mathsf{i}' \quad \Rightarrow \quad \mathsf{PL}(\mathsf{i}') \neq \epsilon \\ \mathsf{P},\mathsf{L} \vdash \mathsf{c}\ \leq_{impl}\ \mathsf{i} \quad \Rightarrow \quad \mathsf{PL}(\mathsf{i}) \neq \epsilon\end{array}}{\vdash \mathsf{P},\mathsf{L}\ \diamond_{sups}}$$

Figure 6. Subclasses, acyclic programs, programs with complete superclasses

In statically typed programming languages, such requirements are checked *all together* at compile-time; in dynamically-typed programming languages they are checked *all together* when (and if) the above expression is executed.

In Java, however, these requirements are checked at *various stages* of execution. Consider for example, execution of the verified expression e_2:

$\mathsf{e}_2 \equiv$ v=**new** FrPhil**;** w=**new** FrPhil**;**
v[FrPhil, FrPhil, Food]**.**think(w)

where the class Phil has been loaded and prepared, but no further class has been loaded. Thus, we have a configuration $\mathsf{e}_2, .., \mathsf{P}_{\mathsf{Ph}}, \epsilon$. Then:

S1 v=**new** FrPhil, attempts to load the class FrPhil; if none is found, or a class circularity is encountered, then LoadErr is thrown; otherwise $\mathsf{L}_{\mathsf{FrPh}}$is loaded, and we continue execution with **new** FrPhil..., .., $\mathsf{P}_{\mathsf{Ph}}, \mathsf{L}_{\mathsf{FrPh}}$.

S2 The verifier checks $\mathsf{L}_{\mathsf{FrPh}}$, and in the process it checks all methods in that class. In order to verify the body of method Food think(FrPhil x) in FrPhil, the verifier needs to establish that Pear is a subclass of Food. For this it tries to load Pear and its superclasses. If these cannot be found, LoadErr is thrown, otherwise **R1** is established. If they can be found, but do not satisfy the subtype requirement, VerifErr is thrown. Otherwise, **R2** is established, class FrPhil is prepared, and a new FrPhil object is created. We continue with v = **new** FrPhil.., .., $\mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{FrPh}}, \mathsf{L}_{\mathsf{Pear}}\mathsf{L}_{\mathsf{Food}}$.[12]

[12]We assumed that Pear is a direct subclass of Food, which is a direct subclass of Object.

S3 A new FrPhil object can now be created and its address assigned to v; we continue with w = **new** FrPhil.., .., $\mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{FrPh}}, \mathsf{L}_{\mathsf{Pear}}\mathsf{L}_{\mathsf{Food}}$

S4 w=**new** FrPhil creates a second FrPhil object and assigns its address to w, and continues with .., .., $\mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{FrPh}}, \mathsf{L}_{\mathsf{Pear}}\mathsf{L}_{\mathsf{Food}}$.

S5 v[FrPhil, FrPhil, Food]**.**think(w) evaluates v and w, resolves the method think in class FrPhil, and continues with e_1, .., $\mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{FrPh}}, \mathsf{L}_{\mathsf{Pear}}\mathsf{L}_{\mathsf{Food}}$.

S6 **new** Pear attempts to verify $\mathsf{L}_{\mathsf{Pear}}$ and $\mathsf{L}_{\mathsf{Food}}$; if unsuccessful it throws VerifErr. Otherwise, it establishes that *any* methods defined in class Pear or inherited from its superclasses will be "well-behaved" (this means that **R5**⇒**R6**). Execution continues with **new** Pear, .., $\mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{FrPh}}\mathsf{P}_{\mathsf{Pear}}\mathsf{P}_{\mathsf{Food}}, \epsilon$.

S7 **new** Pear creates a Pear object at some address α, and continues with α, .., $\mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{FrPh}}\mathsf{P}_{\mathsf{Pear}}\mathsf{P}_{\mathsf{Food}}, \epsilon$.

S8 **new** Salt attempts to load class Salt; if unsuccessful, it throws LoadErr; otherwise it continues with **new** Salt, .., $\mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{FrPh}}\mathsf{P}_{\mathsf{Pear}}\mathsf{P}_{\mathsf{Food}}, \mathsf{L}_{\mathsf{Salt}}$.

S9 $\mathsf{L}_{\mathsf{Salt}}$ is verified; if unsuccessful, then VerifErr is thrown; otherwise **R3** and **R4** have been established, and we continue with **new** Salt, .., $\mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{FrPh}}\mathsf{P}_{\mathsf{Pear}}\mathsf{P}_{\mathsf{Food}}\mathsf{P}_{\mathsf{Salt}}, \epsilon$.

S10 a Salt object is created at some address α'; execution continues with
α[Food, Salt, Food]**.**cook(α'), .., $\mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{FrPh}}\mathsf{P}_{\mathsf{Pear}}\mathsf{P}_{\mathsf{Food}}\mathsf{P}_{\mathsf{Salt}}, \epsilon$.

S11 α[Food, Salt, Food].cook(α') attempts to resolve the method cook with parameter Salt and result type Food in class Food. If unsuccessful, it throws NoMethErr. Otherwise, **R5** has been established, which, together with **R5⇒R6** from **S4** establishes **R6**, and execution continues with the appropriate method body.

In the above example we see that execution of verified code might throw verification, loading, or resolution errors. Thus, verification alone does not ensure "well-behavedness".

On the other hand, as shown in section 3.4, execution of unverified expression e_3[13]

$e_3 \equiv$ aPh[FrPhil, Food].like = **new** Pear

in configuration $e_3, \sigma_0, P_{Ph}P_{FrPh}, \epsilon$ (for σ_0 from section 3) leads to configuration $12, \sigma_1, P_{Ph}P_{FrPh}, \epsilon$, where $\sigma_1 = \sigma_0[8 \mapsto 12, 12 \mapsto \text{Pear}, ..]$. In the new store, σ_1, the class of the object at address 8 has been overwritten by an address; the consistency of the store has been destroyed! Thus, resolution checks alone do not ensure "well-behavedness" either.

Notice also, that **R3** and **R4** are *not* attempted in stage **S1** – more in section 3.6.

In the appendix we give an example which demonstrates the treatment of interfaces based on the one given by Buechi[2]. We now study the five components of execution:

3.3 Evaluation

Evaluation is the part of execution that is not affected by dynamic linking and verification. It is described in the first section of figure 4, and it comprises:

- propagation, *i.e.* propagate execution at the receiver and then the argument of a method call, at the receiver of a field access and to the left hand and right hand sides of an assignment (rule PROPAGATE), [14],
- throwing the NllPErr exception when attempting to call a method, access a field, or assign to a field of **0** (rule NULLPOINTERR),
- accessing variables or addresses (ACC), and assigning to variables (VARASS)
- creating new objects (NEW) of already prepared class c (*i.e.* $P(c) \neq \epsilon$), and initializing the fields with **0** at the offsets prescribed in P. (The function $\mathcal{F}s(c, P)$, defined in figure 12, returns types and offsets for all fields declared in class c or in any of c's superclasses.)

[13] The expression e_3 could be the result of a compilation of expression $e_4 \equiv$ aPh.like = **new** Pear where aPh had been declared of type FrPhil, and the type of aPh was modified without re-compiling e_4. This could happen if aPh stood for a method parameter, or a field in another class.

[14] For the sake of succinctness we did not supply rules for the propagation of exceptions; these would have been standard.

3.4 Resolution

Resolution describes the process of resolving references to fields or methods. It corresponds to the bytecode instructions getfield, putfield, invokeinterface and invokevirtual.

In Java implementations, resolution may also take place during linking. The related exceptions, NoMethErr and NoFldErr, *could* be anticipated at link time; indeed the language specification leaves some leeway, and requires that linkage-related exceptions may only be thrown when an action is taken that might require linkage to the class or interface involved in the error, *c.f.* 12.1.2 of [12]. Our model follows the laziest possible approach as to the timing of throwing link-related exceptions, which also coincides with current implementations. [15]

3.4.1 Field Resolution

Field access has the form $\alpha[t_1, t_2]$.f. The offset of that field is determined using $\mathcal{F}o(f, t_1, t_2, P)$. This function, defined in figure 12, searches the class hierarchy for a definition of a field f with type t_2, starting with class t_1 and continuing with the superclasses. If the offset is found, *i.e.* $\mathcal{F}o(f, t_1, t_2, P) = \beta$, then it is used to calculate the address of that field, *i.e.* $\alpha + \beta$ (FLDACC1). Thus, our model describes address calculations, and is, in that sense, at a lower-level than those in [10, 20, 19].

If t_1 is defined, but does not have a declaration for field f of type t_2, *i.e.* $\mathcal{F}o(f, t_1, t_2, P) = \mathbf{-1}$, or if t_1 is an interface, *i.e.* $\mathcal{F}o(f, t_1, t_2, P) = \mathbf{-2}$, then exceptions are thrown (FLDACC2,FLDACC3). Note, that the case where $\mathcal{F}o(f, t_1, t_2, P) = \mathbf{-3}$ need not be treated here, as it corresponds to the case where t_1 has not been prepared yet, and it is treated by the rules for loading, verification and preparation, ie LOADDERR, VERIFERR, LOADPREPVERIF.

The offset calculation $\mathcal{F}o(f, t_1, t_2, P)$ uses the stored, *static* type t_1, and not the actual, *dynamic* class of the object in α. This is why the configuration aPh[FrPhil, Food].like = **new** Pear, $\sigma_0, P_{Ph}P_{FrPh}, \epsilon$ leads to the unsafe configuration $12, \sigma_1, P_{Ph}P_{FrPh}, \epsilon$ described in section 3.2.[16]!

Such problems do not arise for previously verified code. For example, the term aPh[FrPhil, Food].like = **new** Pear would not verify, because the type of aPh is not a subtype of FrPhil, In general, as we shall see later, in well formed code P, the offset $\mathcal{F}o(f, t_1, t_2, P)$ represents the position for field f with type t_2 inherited from class t_1 for all objects of class t_1 or *any* subclass of t_1.[17] Provided that the field access has been checked by the verifier, and thus that the

[15] A more general model, reflecting this leeway through restricted non-determinism in the operational semantics, could be tackled in future research.

[16] because $\mathcal{F}o(\text{like}, \text{FrPhil}, \text{Food}, P_{FrPh}P_{Ph}) = 3$.

[17] *c.f.* the last rule in figure 9.

type of α is indeed a subclass of t_1, the address calculation will return the appropriate field stored in the object at α.

3.4.2 Method Call Resolution

Method calls have the form $\alpha[\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3].\mathsf{m}(\gamma)$. The offset is determined by the function $\mathcal{M}o(\mathsf{m},\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3,\mathsf{P})$, defined in figure 12. This function considers m, the name of the method, t_1, the class containing the method, t_2, the type of the argument, and t_3, the result. The latter two are necessary for overloading resolution. If t_1 is an interface, then $\mathcal{M}o(\mathsf{m},\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3,\mathsf{P})$= **-2**, and the exception ClssChngErris thrown[18]. If class t_1 exists, but no such method can be found in t_1, the exception NoMethErr is thrown (METHCALL2).

As for fields, the *actual class* of the receiver, *i.e.* the class of α, is *not* considered. If a method is found, *i.e.* if $\mathcal{M}o(\mathsf{m},\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3,\mathsf{P})$=$\beta$ for some β, then β is used to select the method body from the lookup table of the class of α through ($\mathcal{M}e(\beta,\sigma(\alpha),\mathsf{P})$ in METHCALL3) – here the actual class of the receiver *is* used. This is so, because, as we shall see, in well-formed P's, corresponding methods have the same offset in the lookup tables of t_1 and in all the subclasses of t_1.[19] Well formedness of the method call (as guaranteed by verification) ensures that the class of α is, indeed, a subclass of t_1.

As for fields, the case where t_1 has not been prepared yet is taken care of by the loading, verification and preparation rules.

3.4.3 Interface Method Call Resolution

Interface method calls have the form $\alpha[\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3]^{\mathbf{i}}.\mathsf{m}(\gamma)$. The method is first looked up in the interface through $\mathcal{M}o^{\mathbf{i}}(\mathsf{m},\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3,\mathsf{P})$. If t_1 is a class[20], or if the class of the receiver, denoted by $\sigma(\alpha)$, does not implement t_1[21] then the exception ClssChngErr is thrown. If interface t_1 exists, but does not contain nor inherit an appropriate method declaration, [22] then the exception NoMethErr. Otherwise, the method is looked up in the *actual class* of the receiver, *i.e.* offset is determined by the function $\mathcal{M}o(\mathsf{m},\sigma(\alpha),\mathsf{t}_2,\mathsf{t}_3,\mathsf{P})$ and then the method body with the corresponding offset is executed (INTFMETHCALL4).

If we compare method calls and interface method calls, we notice that the latter require the extra check. Namely INTFMETHCALL3 ascertains that the receiver implements t_1. Such a check is not necessary for method calls, $\mathsf{e}'_1[\mathsf{t}'_1,\mathsf{t}'_2,\mathsf{t}'_3].\mathsf{m}(\mathsf{e}'_2)$, because verification guarantees that e'_1 will evaluate to an object of a subtype of t'_1. However, the verifier is more lenient with interface method calls, and verification of $\mathsf{e}_1[\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3]^{\mathbf{i}}.\mathsf{m}(\mathsf{e}_2)$ does not guarantee that e_1 will evaluate to an object of a subtype of t_1; therefore this needs to be checked at the time of execution of the method call.

As for fields and for method calls, the case where t_1 has not been prepared yet is taken care of by the loading, verification and preparation rules.

3.5 Loading

Loading is required when a type context, $\sqsubset \mathsf{t} \sqsupset^{typ}$, is executed for a class/interface t which has not been loaded yet. That is, when a new object of class t is created, or a when a field of class t is accessed, or when a method from class or interfeace t is called.

If loading is successful, *i.e.* $ld(\mathsf{t},\mathsf{P},\mathsf{L}) = \mathsf{L}' \neq \epsilon$, then execution continues with the loaded code augmented by L' (LOAD), otherwise an exception is thrown (LOADERR).

A *loader* function $ld(\mathsf{t},\mathsf{P},\mathsf{L})$ returns class or interface definitions for t and all its superclasses and superinterfaces except for those already defined in P or L, provided that no class or interface circularity was encountered; otherwise it returns ϵ. Any function satisfying the requirements from figure 11, is a loader. A "real" loader would lookup class definitions in the filesystem or a database, which may be modified from outside the Java program, and so different calls of the loader for the same class might return different binaries. Rather than providing a filesystem/database parameter, in our model different loader functions may be called, thus giving the same effect.

We have taken a simplified view of loading, and have disregarded the possibility of class de-allocation and multiple loaders implementing different search strategies, which we shall consider in future research.

3.6 Verification

Verification is required when executing a type context $\sqsubset \mathsf{t} \sqsupset^{typ}$, and t has been loaded but not yet prepared, *i.e.* $\mathsf{P}(\mathsf{t}) = \epsilon \neq \mathsf{L}_1\mathsf{L}_2(\mathsf{t})$. The loaded code consists of L_1 and L_2, where L_1 is the part of the loaded code which contains the definition of t and its supertypes, except for those already defined in P, *i.e.* $\mathsf{L}_1(\mathsf{t}) \neq \epsilon$, and $\vdash \mathsf{P},\mathsf{L}_1 \diamond_{sups}$. Then L_1 is verified. If verification succeeds and requires

[18] This can happen, if one compiles t_1 as a class, then compiles the class containing the method call, then re-compiles t_1 as an interface, and does not re-compile the class with the method call.

[19] *c.f.* the last rule in figure 9.

[20] This can happen, if one compiles t_1 as an interface, then compiles the class containing the method call, then recompiles t_1 as a class, and does not re-compile the class with the method call.

[21] This can happen, if one compiles a class c', a superclass of $\sigma(\alpha)$ while c' implements the interface t_1, then compiles the class containing the method call, then recompiles making sure that none of the superclasses of $\sigma(\alpha)$ implement the interface t_1, and does not re-compile the method call.

[22] This can happen, if one compiles t_1 with the method declaration, then compiles the method call, then removes from t_1 the method declaration, and re-compiles t_1 but does not re-compile the method call.

$$\frac{\vdash\ \mathsf{P,L}\ \Diamond_a}{\mathsf{P,L} \vdash_v \mathsf{t} \ \leq\ \mathsf{t} \quad \underset{\mathit{loads}}{\leftharpoonup} \epsilon} \tag{1}$$

$$\frac{\begin{array}{l}\vdash\ \mathsf{P,L}\ \Diamond_a \\ \mathsf{P,L} \vdash \mathsf{c}\ \leq_{clss}\ \mathsf{c'}\end{array}}{\mathsf{P,L} \vdash_v \mathsf{c} \ \leq\ \mathsf{c'} \quad \underset{\mathit{loads}}{\leftharpoonup} \epsilon} \tag{2}$$

$$\frac{\begin{array}{l}\vdash\ \mathsf{P,L}\ \Diamond_a \\ \mathsf{PL(i)} = \textbf{interface} \ldots\end{array}}{\mathsf{P,L} \vdash_v \mathsf{t} \ \leq\ \mathsf{i} \quad \underset{\mathit{loads}}{\leftharpoonup} \epsilon} \tag{3}$$

$$\frac{\vdash\ \mathsf{P,L}\ \Diamond_a}{\mathsf{P,L} \vdash_v \textbf{int} \ \leq\ \textbf{int} \quad \underset{\mathit{loads}}{\leftharpoonup} \epsilon} \tag{4}$$

$$\frac{\begin{array}{l}\vdash\ \mathsf{P,L}\ \Diamond_a \\ \mathsf{PL(c)} = \epsilon \\ ld(\mathsf{c,P,L}) = \mathsf{L'} \\ \mathsf{P,LL'} \vdash \mathsf{c}\ \leq_{clss}\ \mathsf{c'}\end{array}}{\mathsf{P,L} \vdash_v \mathsf{c} \ \leq\ \mathsf{c'} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'}} \tag{5}$$

$$\frac{\begin{array}{l}\vdash\ \mathsf{P,L}\ \Diamond_a \\ \mathsf{PL(i)} = \epsilon,\ ld(\mathsf{i,P,L}) = \mathsf{L'} \\ \mathsf{L'(i)} = \textbf{interface} \ldots\end{array}}{\mathsf{P,L} \vdash_v \mathsf{t} \ \leq\ \mathsf{i} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'}} \tag{6}$$

$$\frac{\vdash\ \mathsf{P,L}\ \Diamond_a}{\begin{array}{l}\mathsf{P,L,E} \vdash_v \gamma\ :\ \mathsf{int} \quad \underset{\mathit{loads}}{\leftharpoonup} \epsilon \\ \mathsf{P,L,E} \vdash_v \mathbf{0}\ :\ \mathsf{c} \quad \underset{\mathit{loads}}{\leftharpoonup} \epsilon \\ \mathsf{P,L,E} \vdash_v \textbf{new}\ \mathsf{c}\ :\ \mathsf{c} \quad \underset{\mathit{loads}}{\leftharpoonup} \epsilon\end{array}} \tag{7}$$

$$\frac{\begin{array}{l}\vdash\ \mathsf{P,L}\ \Diamond_a \\ \mathsf{E(y)} = \mathsf{t}\end{array}}{\mathsf{P,L,E} \vdash_v \mathsf{y}\ :\ \mathsf{t} \quad \underset{\mathit{loads}}{\leftharpoonup} \epsilon} \tag{8}$$

$$\frac{\begin{array}{l}\mathsf{P,L,E} \vdash_v \mathsf{var}\ :\ \mathsf{t} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'} \\ \mathsf{P,LL',E} \vdash_v \mathsf{e}\ :\ \mathsf{t'} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L''} \\ \mathsf{P,LL'L''} \vdash_v \mathsf{t'}\ \leq\ \mathsf{t} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'''}\end{array}}{\mathsf{P,L,E} \vdash_v \mathsf{var = e}\ :\ \mathsf{t'} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'L''L'''}} \tag{9}$$

$$\frac{\begin{array}{l}\mathsf{P,L,E} \vdash_v \mathsf{e}\ :\ \mathsf{t} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'} \\ \mathsf{P,LL'} \vdash_v \mathsf{t}\ \leq\ \mathsf{t_1} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L''}\end{array}}{\mathsf{P,L,E} \vdash_v \mathsf{e[t_1,t_2].f}\ :\ \mathsf{t_2} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'L''}} \tag{10}$$

$$\frac{\begin{array}{l}\mathsf{P,L,E} \vdash_v \mathsf{e_1}\ :\ \mathsf{t'_1} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_1} \\ \mathsf{P,LL'_1,E} \vdash_v \mathsf{e_2}\ :\ \mathsf{t'_2} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_2} \\ \mathsf{P,LL'_1L'_2} \vdash_v \mathsf{t'_1}\ \leq\ \mathsf{t_1} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_3} \\ \mathsf{P,LL'_1L'_2L'_3} \vdash_v \mathsf{t'_2}\ \leq\ \mathsf{t_2} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_4}\end{array}}{\mathsf{P,L,E} \vdash_v \mathsf{e_1[t_1,t_2,t_3].m(e_2)}\ :\ \mathsf{t_3} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_1L'_2L'_3L'_4}} \tag{11}$$

$$\frac{\begin{array}{l}\mathsf{P,L,E} \vdash_v \mathsf{e_1}\ :\ \mathsf{t'_1} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_1} \\ \mathsf{P,LL'_1,E} \vdash_v \mathsf{e_2}\ :\ \mathsf{t'_2} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_2} \\ \mathsf{P,LL'_1L'_2} \vdash_v \mathsf{t'_2}\ \leq\ \mathsf{t_2} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_4}\end{array}}{\mathsf{P,L,E} \vdash_v \mathsf{e_1[t_1,t_2,t_3]^i.m(e_2)}\ :\ \mathsf{t_3} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_1L'_2L'_4}} \tag{12}$$

$$\frac{\begin{array}{ll}\mathsf{PL(c')} = \textbf{class} \ldots & \\ \mathsf{PL(i)} = \textbf{interface} \ldots & \\ \mathsf{f_i = f_j} \ \Rightarrow\ \mathsf{i = j} & 1\leq \mathsf{i,j} \leq \mathsf{n} \\ \mathsf{m_i = m_j}\ \text{ and }\ \mathsf{t_{i1} = t_{j1}}\ \text{ and }\ \mathsf{t_{i2} = t_{j2}} \ \Rightarrow\ \mathsf{i = j} & 1\leq \mathsf{i,j} \leq \mathsf{k} \\ \mathsf{P,LL'_1 \ldots L'_{2(i-1)}, (t_{i2}\ x, c\ \textbf{this})} \vdash_v \mathsf{e_i}\ :\ \mathsf{t'_{i1}} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_{2i-1}} & 1\leq \mathsf{i} \leq \mathsf{k} \\ \mathsf{P,LL'_1 \ldots L'_{2i-1}} \vdash_v \mathsf{t'_{i1}}\ \leq\ \mathsf{t_{i1}} \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_{2i}} & 1\leq \mathsf{i} \leq \mathsf{k}\end{array}}{\mathsf{P,L} \vdash_v \textbf{class}\ \mathsf{c}\ \textbf{ext}\ \mathsf{c'}\ \textbf{impl} \ldots\mathsf{i}\ldots\{\ \mathsf{t_1\ f_1} \ldots \mathsf{t_n\ f_n} \quad \mathsf{t_{11}\ m_1(t_{12}\ x)\{e_1\}} \ldots \mathsf{t_{k1}\ m_k(t_{k2}\ x)\{e_k\}}\ \}\ \Diamond \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_1} \ldots \mathsf{L'_{2k}}} \tag{13}$$

$$\frac{\begin{array}{ll}\mathsf{PL(i')} = \textbf{interface} \ldots & \\ \mathsf{m_i = m_j}\ \text{ and }\ \mathsf{t_{i1} = t_{j1}}\ \text{ and }\ \mathsf{t_{i2} = t_{j2}} \ \Rightarrow\ \mathsf{i = j} & 1\leq \mathsf{i,j} \leq \mathsf{k}\end{array}}{\mathsf{P,L} \vdash_v \textbf{interface}\ \mathsf{i}\ \textbf{ext} \ldots\mathsf{i'}\ldots\{\ \mathsf{t_{11}\ m_1(t_{12}\ x)},\ \ldots,\ \mathsf{t_{k1}\ m_k(t_{k2}\ x)}\ \}\ \Diamond \quad \underset{\mathit{loads}}{\leftharpoonup} \epsilon} \tag{14}$$

$$\frac{\begin{array}{ll}\{\mathsf{t_1}, \ldots \mathsf{t_n}\} = \{\mathsf{t} \mid \mathsf{L'(t)} \neq \epsilon\} & \\ \mathsf{P,LL'_1 \ldots L'_{i-1}} \vdash_v \mathsf{L'(t_i)}\ \Diamond \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_i} & 0\leq \mathsf{i} \leq \mathsf{n}\end{array}}{\mathsf{P,L} \vdash_v \mathsf{L'}\ \Diamond \quad \underset{\mathit{loads}}{\leftharpoonup} \mathsf{L'_1} \ldots \mathsf{L'_n}} \tag{15}$$

Figure 7. Verification

$$Env \quad ::= \quad \epsilon \mid Env,\, type\ z \mid Env,\, type\ \textbf{this} \qquad \frac{\mathsf{E(z)} \neq \epsilon \quad \Rightarrow \quad \mathsf{E'(z)} = \mathsf{E(z)}}{\vdash\ \mathsf{E'}\ \leq \mathsf{E}}$$

Figure 8. Environments

the loading of $\mathsf{L'}$, then L_1 is prepared, and execution continues with the augmented prepared code P_1, and additional loaded code $\mathsf{L'}$, *c.f.* VERIFANDPREP. If verification fails, an exception is thrown, *c.f.* VERIFERR.

Verification in our paper corresponds to the third pass of the "real" verifier as described in ch. 4.9.1 of [18], and is expressed through the judgment

$$\mathsf{P,L} \vdash_v \mathsf{L''} \diamond \quad \underset{loads}{\leftharpoondown} \ \mathsf{L'}$$

meaning that the binary $\mathsf{L''}$ could be verified in the context of the prepared code P, and the loaded but not yet prepared code L, and caused $\mathsf{L'}$ to be loaded (but not verified). Thus, this judgment has the "side-effect" of loading $\mathsf{L'}$.

Verification of classes is defined in terms of verification of expressions, with the judgment

$$\mathsf{P,L,E} \vdash_v \mathsf{e}\ :\ \mathsf{t} \quad \underset{loads}{\leftharpoondown} \ \mathsf{L'}$$

meaning that the expression e could be verified as having type t, in the context of P, L, and the environment E, and caused further classes/interfaces $\mathsf{L'}$ to be loaded (but not verified). This is described in figure 7.

Establishing the above sometimes requires a judgment

$$\mathsf{P,L} \vdash_v \mathsf{t}\ \leq\ \mathsf{t'} \quad \underset{loads}{\leftharpoondown} \ \mathsf{L'}$$

meaning that type t could be verified as widening to type $\mathsf{t'}$ in the context of P and L, and caused further classes/interfaces $\mathsf{L'}$ to be loaded (but not verified). Classes or interfaces may be loaded when trying to establish whether a t, undefined in P or L is a subtype $\mathsf{t'}$, as in rules (5) and (6) of figure 7.

For example, verification of

$\mathsf{e}_1 \equiv$ **new** $\mathsf{Pear[Food, Salt, Food]}$**.**$\mathsf{cook(}$ **new** $\mathsf{Salt)}$

requires establishing that Pear widens to Food, which, in its turn, if Pear is not loaded, requires loading Pear and all its superclasses. Therefore, if

$ld(\mathsf{Pear}, \mathsf{P_{Ph}}, \epsilon) = \mathsf{L_{Pear}L_{Food}}$,

and the superclass of Pear is Food, then:

$\mathsf{P_{Ph}}, \epsilon \vdash_v \mathsf{Pear}\ \leq\ \mathsf{Food} \quad \underset{loads}{\leftharpoondown} \mathsf{L_{Pear}L_{Food}}$.

The difference between (5) and (6) is, that in (5) c and all its superclasses are loaded, whereas in (6) only i is loaded.

The assertion $\mathsf{P,L} \vdash_v \mathsf{t}\ \leq\ \mathsf{t} \quad \underset{loads}{\leftharpoondown} \epsilon$ holds for any t, *c.f.* rule (1). Thus, verification assumes *any* identifier to stand for a class, or interface and so to widen to itself. Therefore,

$\mathsf{P_{Ph}}, \epsilon \vdash_v \mathsf{Salt}\ \leq\ \mathsf{Salt} \quad \underset{loads}{\leftharpoondown} \epsilon$.

Also, the assertion $\mathsf{P,L} \vdash_v \mathsf{t}\ \leq\ \mathsf{i} \quad \underset{loads}{\leftharpoondown} \epsilon$ holds for any interface i, *c.f.* rules (3) and (6). Thus verification assumes *any* identifier to widen to i, provided that i stands for an already loaded or prepared interface.

Verification is "optimistic" with respect to method calls and field accesses (rules (11) and (12)), and more liberal than the Java source checks. For field access, $\mathsf{e}_1[\mathsf{t}_1,\mathsf{t}_2].\mathsf{f}$, verification only checks that the type of e_1 widens to t_1, the static type in the signature, and gives to the whole expression the type t_2 – it does *not* attempt to check the existence of a field with type t_2, but leaves this to the resolution checks. Similarly for method calls. Therefore, verification of e_1 will load Food and Pear, and not Salt, and will not verify either of these classes, *i.e.*

$\mathsf{P_{Ph}}, \epsilon, \epsilon \vdash_v \mathsf{e}_1\ :\ \mathsf{Food} \quad \underset{loads}{\leftharpoondown} \mathsf{L_{Pear}L_{Food}}$

Verification of a class (rule (13)) does not imply verification of all classes used: Even though $\mathsf{L_{Ph}}$ mentions the classes FrPhil, Book, Food, and Pear, its verification only requires class Pear and all its superclasses to be loaded. Thus,

$\epsilon, \mathsf{L_{Ph}} \vdash_v \mathsf{L_{Ph}} \diamond \quad \underset{loads}{\leftharpoondown} \mathsf{L_{Food}L_{Pear}}$.

Finally, if an order can be found to verify classes and/or interfaces $\mathsf{t_i}$, then verification is successful, *c.f.* rule (15).

Verification requires type assignments, expressed through an *environment*, E, which is a sequence of declarations of the form $\mathsf{t_i}\ \mathsf{var_i}$. Environments are declared in figure 8; they should contain unique declarations, as expressed by the judgment $\vdash\ \mathsf{E}\ \diamond_u$, and allow looking up the type of variable z through $\mathsf{E(z)}$.[23] We do *not* require the $\mathsf{t_i}$ to indicate types declared in P or L. So, an environment may use identifiers as types which have no corresponding definition in P or L.

In summary, verification is only concerned with widening, but not with the existence of fields or methods. Nor does verification enforce the Java source rules forbidding methods with same identifier and argument types, but different result types. In our example class Phil defines a method `Book think( FrPhil x){...}` and FrPhil defines a method `Food think( FrPhil x){...}`. Though illegal Java source, it is legal bytecode.

[23] We do not define $\vdash\ \mathsf{E}\ \diamond_u$, nor $\mathsf{E(z)}$, because they are standard.

3.7 Preparation

If verification is successful, the corresponding binaries are prepared through the function $pr : \mathcal{P} \times \mathcal{L} \longrightarrow \mathcal{P}$, which maps binary L to $pr(\mathsf{P}, \mathsf{L})$ using information from P. Preparation is concerned with determining the object layout (through adding offsets to fields), and with creating the method look-up table (through copying some methods from superclasses, and allocating offsets to method bodies).

Rather than prescribe the exact strategy for offset determination, we give requirements in figure 11, *i.e.* a mapping is a *linker* if it allocates distinct offsets, copies from the superclass all non-overridden methods with their offsets unaltered, allocates to overriding methods the offset from the corresponding overridden method, and allocates fresh offsets to the remaining methods. For the example from figure 2, a linker pr_0, allocating consecutive offsets, would give $pr_0(\epsilon, \mathsf{L}_{\mathsf{Ph}})$=$\mathsf{P}_{\mathsf{Ph}}$, $pr_0(\mathsf{P}_{\mathsf{Ph}}, \mathsf{L}_{\mathsf{FrPh}})$=$\mathsf{P}_{\mathsf{FrPh}}$, and $pr_0(\epsilon, \mathsf{L}_{\mathsf{Ph}}\mathsf{L}_{\mathsf{FrPh}})$=$\mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{FrPh}}$.

4 Soundness

A subject reduction theorem demonstrates that the Java approach described here indeed preserves types. For this we first define what it means for prepared code P to be well-formed, and for a state σ to conform to P and E.

4.1 Well formed prepared code

The judgment $\mathsf{L} \vdash \mathsf{P} \diamond$, defined in figure 9, guarantees that the prepared code P is well formed in the context of loaded code L. The main requirements for well-formedness of prepared code are:

- all classes/interfaces defined in P have their superclasses/superinterfaces in P,
- fields defined in a class c have the same offset in all subclasses of c,
- methods defined in a class c have the same offset in all subclasses of c,
- method bodies are well-formed and respect their signatures.

As in verification, well-formedness of prepared code does not guarantee the existence of any fields or methods required in method bodies.

In contrast to verification, well-formedness of linked code does not cause loading of further binaries. Also, while judgment $\mathsf{P}, \mathsf{L} \vdash_{\!\!v} \mathsf{L}' \diamond \ \underset{loads}{\leftharpoondown} \ \mathsf{L}''$ represents checks that are performed by Java implementations, the judgment $\mathsf{L} \vdash \mathsf{P} \diamond$ is only a vehicle for proving soundness.

However, the criteria for well-formedness of P can give us an intuition as to why the Java approach works, and also, ideas about alternative approaches.

4.2 Conformance and run-time types

The judgment $\mathsf{P}, \mathsf{E} \vdash_{\!\!c} \sigma \diamond$, defined in figure 10, expresses that the store σ *conforms* to prepared program P and to variable declarations in E. The main requirements are

- all classes/interfaces defined in P have their superclasses/superinterfaces in P,
- the classes of all objects stored in σ are defined in P,
- all objects stored in σ contain appropriate values at the offsets of the fields of their class,
- no object is stored inside another object,
- all variables defined in E have in σ values appropriate to their types,
- an object of class c is an appropriate value for any superclass of c, and it is an appropriate value for any interface.

The judgment $\sigma, \mathsf{P} \vdash_c \alpha \diamond$ expresses that the address α points to an object of some class c, which contains at the corresponding offsets appropriate values for all fields of c. In order to obtain a well-founded relation, we defined conformance in terms of the auxiliary *weak conformance* judgment $\sigma, \mathsf{P} \vdash_{cw} \gamma : \mathsf{t}$. Notice, that a positive value γ may conform to both **int** and a class type, and to *any* interface type, *e.g.* $\sigma_0, \mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{Food}} \vdash_{cw} 5 : \mathbf{int}$, $\sigma_0, \mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{Food}} \vdash_{cw} 5 : \mathsf{Phil}$, but $\sigma_0, \mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{Food}} \not\vdash_{cw} 5 : \mathsf{Food}$. Also, if $\mathsf{P}_{\mathsf{BankIntf}}$ contains the declaration of an interface $\mathsf{BankIntf}$, then $\sigma_0, \mathsf{P}_{\mathsf{Ph}}\mathsf{P}_{\mathsf{Food}}\mathsf{P}_{\mathsf{BankIntf}} \vdash_{cw} 5 : \mathsf{BankIntf}$.

Notice also, that store conformance does not take the loaded, not yet verified binaries L into account. Also, **0** conforms to any class, allowing objects with a field initialized to **0**, belonging to a yet undefined class. The requirement $\forall \beta' \leq \beta$: $\sigma(\alpha + \beta') \neq \mathsf{c}'$ ensures that no object is stored "inside" another object. It is used to prove that evaluation does not affect the type of expressions (lemma 4).

Types for run-time expressions are given by the judgment $\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{\!\!r} \sigma, \mathsf{e} : \mathsf{t}$, defined in figure 10. The rules are similar to verification, with the difference that for run-time expressions the store σ is taken into account, and that loading of further binaries is not considered.

Typing uses the widening judgment $\mathsf{P}, \mathsf{L} \vdash \mathsf{t}' \leq \mathsf{t}$, from figure 10, expressing that t' can be widened to t using the information from the prepared program P and the loaded program L. [24]

[24] We can prove that $\mathsf{P}, \mathsf{L} \vdash \mathsf{t}' \leq \mathsf{t}$ iff $\mathsf{P}, \mathsf{L} \vdash_{\!\!v} \mathsf{t}' \leq \mathsf{t} \ \underset{loads}{\leftharpoondown} \ \epsilon$.

$$\frac{\vdash\ \mathsf{P,L}\ \diamond_a}{\mathsf{P,L} \vdash \mathsf{c}\ \leq\ \mathsf{c}} \qquad \frac{\vdash\ \mathsf{P,L}\ \diamond_a}{\mathsf{P,L} \vdash \mathbf{int}\ \leq\ \mathbf{int}} \qquad \frac{\begin{array}{l}\vdash\ \mathsf{P,L}\ \diamond_a \\ \mathsf{P,L} \vdash \mathsf{c}\ \leq_{clss}\ \mathsf{c}'\end{array}}{\mathsf{P,L} \vdash \mathsf{c}\ \leq\ \mathsf{c}'} \qquad \frac{\begin{array}{l}\vdash\ \mathsf{P,L}\ \diamond_a \\ \mathsf{PL(i)} = \mathbf{interface}\ ...\end{array}}{\mathsf{P,L} \vdash \mathsf{t}\ \leq\ \mathsf{i}}$$

$$\frac{\begin{array}{l}\vdash\ \mathsf{P,L}\ \diamond_a \\ \mathsf{E(y)} = \mathsf{t}\end{array}}{\begin{array}{l}\mathsf{P,L,E} \vdash \gamma\ :\ \mathsf{int} \\ \mathsf{P,L,E} \vdash \mathbf{0}\ :\ \mathsf{c} \\ \mathsf{P,L,E} \vdash \mathbf{new}\ \mathsf{c}\ :\ \mathsf{c} \\ \mathsf{P,L,E} \vdash \mathsf{y}\ :\ \mathsf{t}\end{array}} \qquad \frac{\begin{array}{l}\mathsf{P,L,E} \vdash \mathsf{var}\ :\ \mathsf{t} \\ \mathsf{P,L,E} \vdash \mathsf{e}\ :\ \mathsf{t}' \\ \mathsf{P,L} \vdash \mathsf{t}'\ \leq\ \mathsf{t}\end{array}}{\mathsf{P,L,E} \vdash \mathsf{var = e}\ :\ \mathsf{t}'} \qquad \frac{\begin{array}{l}\mathsf{P,L,E} \vdash \mathsf{e}\ :\ \mathsf{t} \\ \mathsf{P,L} \vdash \mathsf{t}\ \leq\ \mathsf{t}_1\end{array}}{\mathsf{P,L,E} \vdash \mathsf{e}[\mathsf{t}_1,\mathsf{t}_2].\mathsf{f}\ :\ \mathsf{t}_2}$$

$$\frac{\begin{array}{l}\mathsf{P,L,E} \vdash \mathsf{e}_1\ :\ \mathsf{t}_1' \\ \mathsf{P,L,E} \vdash \mathsf{e}_2\ :\ \mathsf{t}_2' \\ \mathsf{P,L} \vdash \mathsf{t}_1'\ \leq\ \mathsf{t}_1 \\ \mathsf{P,L} \vdash \mathsf{t}_2'\ \leq\ \mathsf{t}_2\end{array}}{\mathsf{P,L,E} \vdash \mathsf{e}_1[\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3].\mathsf{m}(\mathsf{e}_2)\ :\ \mathsf{t}_3} \qquad \frac{\begin{array}{l}\mathsf{P,L,E} \vdash \mathsf{e}_1\ :\ \mathsf{t}_1' \\ \mathsf{P,L,E} \vdash \mathsf{e}_2\ :\ \mathsf{t}_2' \\ \mathsf{P,L} \vdash \mathsf{t}_2'\ \leq\ \mathsf{t}_2\end{array}}{\mathsf{P,L,E} \vdash \mathsf{e}_1[\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3]^{\mathbf{i}}.\mathsf{m}(\mathsf{e}_2)\ :\ \mathsf{t}_3}$$

$$\frac{\begin{array}{ll}\mathsf{P(c')} = \mathbf{class}\ ... & \\ \mathsf{P(i)} = \mathbf{interface}\ ... & \\ \mathsf{f_i = f_j}\ \text{and}\ \mathsf{t_i = t_j} \Rightarrow \mathsf{i = j} & 1 \leq \mathsf{i,j} \leq \mathsf{n} \\ \mathsf{m_i = m_j}\ \text{and}\ \mathsf{t_{i1} = t_{j1}}\ \text{and}\ \mathsf{t_{i2} = t_{j2}} \Rightarrow \mathsf{i = j} & 1 \leq \mathsf{i,j} \leq \mathsf{k} \\ \mathsf{P,L,(t_{i2}\ x, c}\ \mathbf{this}) \vdash \mathsf{e_i}\ :\ \mathsf{t}_{i1}' & 1 \leq \mathsf{i} \leq \mathsf{k} \\ \mathsf{P,L} \vdash \mathsf{t}_{i1}'\ \leq\ \mathsf{t}_{i1} & 1 \leq \mathsf{i} \leq \mathsf{k} \\ \mathcal{F}o(\mathsf{f,c',t,P}) > 0 \Rightarrow \mathcal{F}o(\mathsf{f,c,t,P}) = \mathcal{F}o(\mathsf{f,c',t,P}) & \text{for all identifiers f,t} \\ \mathcal{M}o(\mathsf{m,c',t,t',P}) > 0 \Rightarrow \mathcal{M}o(\mathsf{m,c,t,t',P}) = \mathcal{M}o(\mathsf{m,c',t,t',P}) & \text{for all identifiers t,t}',\ \mathsf{m} \\ \beta_i = \beta_j \Rightarrow \mathsf{i = j} & 1 \leq \mathsf{i,j} \leq \mathsf{n} \\ \beta_i' = \beta_j' \Rightarrow \mathsf{i = j} & 1 \leq \mathsf{i,j} \leq \mathsf{k}\end{array}}{\mathsf{P,L} \vdash \mathbf{class}\ \mathsf{c}\ \mathbf{ext}\ \mathsf{c}'\ \mathbf{impl}\ ...\mathsf{i}...\{\ \mathsf{t_1\ f_1}\ \beta_1\ ...\ \mathsf{t_n\ f_n}\ \beta_n\quad \mathsf{t_{11}\ m_1(t_{12}\ x)\{e_1\}}\ \beta_1'\ ...\ \mathsf{t_{k1}\ m_k(t_{k2}\ x)\{e_k\}}\ \beta_k'\ \}\diamond}$$

$$\frac{\begin{array}{ll}\mathsf{P(i')} = \mathbf{interface}\ ... & \\ \mathsf{m_i = m_j}\ \text{and}\ \mathsf{t_{i1} = t_{j1}}\ \text{and}\ \mathsf{t_{i2} = t_{j2}} \Rightarrow \mathsf{i = j} & 1 \leq \mathsf{i,j} \leq \mathsf{k}\end{array}}{\mathsf{P,L} \vdash \mathbf{interface}\ \mathsf{i}\ \mathbf{ext}\ ...\mathsf{i}'...\{\mathsf{t_{11}\ m_1(t_{12}\ x)}\ ...\ \mathsf{t_{k1}\ m_k(t_{k2}\ x)}\ \}\diamond}$$

$$\frac{\begin{array}{l}\vdash\ \mathsf{P,L}\ \diamond_a \\ \mathsf{P(t)} \neq \epsilon \Rightarrow \mathsf{P,L} \vdash \mathsf{P(t)}\diamond\end{array}}{\mathsf{L} \vdash \mathsf{P}\diamond}$$

Figure 9. Well-formed prepared code

$$\frac{\sigma(\alpha) \text{ an integer value}}{\sigma, \mathsf{P} \vdash_{cw} \alpha : \mathbf{int}} \qquad \frac{\begin{array}{l}\sigma(\alpha) = \mathsf{c}' \\ \mathsf{P}, \epsilon \vdash \mathsf{c}' \ \leq_{clss} \ \mathsf{c}\end{array}}{\sigma, \mathsf{P} \vdash_{cw} \alpha : \mathsf{c}} \qquad \frac{\begin{array}{l}\sigma(\alpha) = \mathsf{c}' \\ \mathsf{P}(\mathsf{i}) = \mathbf{interface} \ ...\end{array}}{\sigma, \mathsf{P} \vdash_{cw} \alpha : \mathsf{i}} \qquad \frac{}{\begin{array}{l}\sigma, \mathsf{P} \vdash_{cw} \mathbf{0} : \mathsf{c} \\ \sigma, \mathsf{P} \vdash_{cw} \gamma : \mathsf{c}\end{array}}$$

$$\frac{\sigma, \mathsf{P} \vdash_{cw} \alpha : \mathbf{int}}{\sigma, \mathsf{P} \vdash_{c} \alpha \diamond} \qquad \frac{\begin{array}{l}\sigma(\alpha) = \mathsf{c} \\ \mathsf{P}(\mathsf{c}) = \mathbf{class} \ ... \\ \forall \mathsf{t} \ \mathsf{f} \ \beta \in \mathcal{F}\mathsf{s}(\mathsf{c}, \mathsf{P}) : \ \ \sigma, \mathsf{P} \vdash_{cw} \sigma(\alpha + \beta) : \mathsf{t}, \\ \qquad \text{and } \forall \beta' \leq \beta : \ \sigma(\alpha + \beta') \neq \mathsf{c}'\end{array}}{\sigma, \mathsf{P} \vdash_{c} \alpha \diamond} \qquad \frac{\begin{array}{l}\vdash \ \mathsf{P}, \epsilon \diamond_{sups} \\ \sigma(\alpha) \neq * \ \Rightarrow \ \sigma, \mathsf{P} \vdash_{c} \alpha \diamond \\ \mathsf{E}(\mathsf{z}) \neq \epsilon \ \Rightarrow \ \sigma, \mathsf{P} \vdash_{cw} \sigma(\mathsf{z}) : \mathsf{E}(\mathsf{z})\end{array}}{\mathsf{P}, \mathsf{E} \vdash_{c} \sigma \diamond}$$

$$\frac{\mathsf{P}, \mathsf{E} \vdash_{c} \sigma \diamond}{\begin{array}{l}\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \gamma \ : \ \mathsf{int} \\ \mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathbf{0} \ : \ \mathsf{c} \\ \mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathbf{new} \ \mathsf{c} \ : \ \mathsf{c}\end{array}} \qquad \frac{\begin{array}{l}\mathsf{P}, \mathsf{E} \vdash_{c} \sigma \diamond \\ \sigma(\alpha) = \mathsf{c} \\ \mathsf{E}(\mathsf{y}) = \mathsf{t}\end{array}}{\begin{array}{l}\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \alpha \ : \ \mathsf{c} \\ \mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{y} \ : \ \mathsf{t}\end{array}} \qquad \frac{\begin{array}{l}\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{var} \ : \ \mathsf{t} \\ \mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{e} \ : \ \mathsf{t}' \\ \mathsf{P}, \mathsf{L} \vdash \mathsf{t}' \ \leq \ \mathsf{t}\end{array}}{\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{var} = \mathsf{e} \ : \ \mathsf{t}'}$$

$$\frac{\begin{array}{l}\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{e} \ : \ \mathsf{t} \\ \mathsf{P}, \mathsf{L} \vdash \mathsf{t} \ \leq \ \mathsf{t}_1\end{array}}{\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{e}[\mathsf{t}_1, \mathsf{t}_2].\mathsf{f} \ : \ \mathsf{t}_2} \qquad \frac{\begin{array}{l}\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{e}_1 \ : \ \mathsf{t}_1' \\ \mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{e}_2 \ : \ \mathsf{t}_2' \\ \mathsf{P}, \mathsf{L} \vdash \mathsf{t}_1' \ \leq \ \mathsf{t}_1 \\ \mathsf{P}, \mathsf{L} \vdash \mathsf{t}_2' \ \leq \ \mathsf{t}_2\end{array}}{\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{e}_1[\mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3].\mathsf{m}(\mathsf{e}_2) \ : \ \mathsf{t}_3} \qquad \frac{\begin{array}{l}\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{e}_1 \ : \ \mathsf{t}_1' \\ \mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{e}_2 \ : \ \mathsf{t}_2' \\ \mathsf{P}, \mathsf{L} \vdash \mathsf{t}_2' \ \leq \ \mathsf{t}_2\end{array}}{\mathsf{P}, \mathsf{L}, \mathsf{E} \vdash_{r} \sigma, \mathsf{e}_1[\mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3]^{\mathsf{i}}.\mathsf{m}(\mathsf{e}_2) \ : \ \mathsf{t}_3}$$

Figure 10. Conformance, and types of runtime expressions

4.3 Locality and preservation of judgments

In general, one expects properties established in a certain context to hold for larger contexts as well. Locality properties were proven in [5], used in [4], and explored in our model of binary compatibility [6].

We can prove the following locality properties: Widening or verification requiring binaries L'_1 and L'_2 to be loaded, only require L'_2 to be loaded, if L'_1 had been loaded before[25]. Also, an expression e with type t in environment E preserves its type in a larger environment E'.

Lemma 1 *For all* P, P_0, L, L_0, L'_1, L'_2, L', e, t, E, F, F':

- $\mathsf{P},\mathsf{L} \vdash_v \mathsf{t} \ \leq\ \mathsf{t}' \quad \underset{loads}{\leftharpoonup} \mathsf{L}'_1\mathsf{L}'_2,$ *and*
  $\vdash\ \mathsf{P}_0\mathsf{P},\mathsf{L}_0\mathsf{L}\mathsf{L}'_1\mathsf{L}'_2\ \diamond_a$
  $\Rightarrow$
  $\mathsf{P}_0\mathsf{P},\mathsf{L}_0\mathsf{L}\mathsf{L}'_1 \vdash_v \mathsf{t}\ \leq\ \mathsf{t}' \quad \underset{loads}{\leftharpoonup} \mathsf{L}'_2$
- $\mathsf{P},\mathsf{L},\mathsf{E} \vdash_v \mathsf{e}\ :\ \mathsf{t} \quad \underset{loads}{\leftharpoonup} \mathsf{L}'_1\mathsf{L}'_2,$ *and*
  $\vdash\ \mathsf{P}_0\mathsf{P},\mathsf{L}_0\mathsf{L}\mathsf{L}'_1\mathsf{L}'_2\ \diamond_a$
  $\Rightarrow$
  $\mathsf{P}_0\mathsf{P},\mathsf{L}_0\mathsf{L}\mathsf{L}'_1,\mathsf{E} \vdash_v \mathsf{e}\ :\ \mathsf{t} \quad \underset{loads}{\leftharpoonup} \mathsf{L}'_2$
- $\mathsf{P},\mathsf{L},\mathsf{E} \vdash_v \mathsf{e}\ :\ \mathsf{t} \quad \underset{loads}{\leftharpoonup} \mathsf{L}',$ *and* $\vdash\ \mathsf{E}'\ \leq \mathsf{E}$
  $\Rightarrow$
  $\mathsf{P},\mathsf{L},\mathsf{E}' \vdash_v \mathsf{e}\ :\ \mathsf{t} \quad \underset{loads}{\leftharpoonup} \mathsf{L}'$

Verification of classes implies verification of the bodies of their methods:

Lemma 2 *For any* P, L, L', L'', c, *if* $\mathsf{P},\mathsf{L} \vdash_v \mathsf{L}''\ \diamond \quad \underset{loads}{\leftharpoonup} \mathsf{L}'$ *and* $\mathsf{L}''(\mathsf{c}) = \textbf{class}\ \mathsf{c}\ \textbf{ext}\ \mathsf{c}'\ \{\ldots\ \mathsf{t}_{i1}\ \mathsf{m}_i(\mathsf{t}_{i2}\ \mathsf{x})\{\ \mathsf{e}_i\ \}\ \ldots\}$, *then, there exist* t'_{i1}, L'_1, L'_2, L'_3 *and* L'_4 *such that* L'=$\mathsf{L}'_1\mathsf{L}'_2\mathsf{L}'_3\mathsf{L}'_4$, *and* $\mathsf{P},\mathsf{L}\mathsf{L}'_1,(\mathsf{t}_{i2}\ \mathsf{x},\mathsf{c}\ \mathbf{this}) \vdash_v \mathsf{e}_i\ :\ \mathsf{t}'_{i1} \quad \underset{loads}{\leftharpoonup} \mathsf{L}'_2$, *and also* $\mathsf{P},\mathsf{L}\mathsf{L}'_1\mathsf{L}'_2 \vdash_v \mathsf{t}'_{i1}\ \leq\ \mathsf{t}_{i1} \quad \underset{loads}{\leftharpoonup} \mathsf{L}'_3$.

Preparation of verified code preserves judgments:

Lemma 3 *For any* P, L_1, L_2, L_3, L', F, e, t, E, σ, *if*

- $\mathsf{P},\mathsf{L}_1\mathsf{L}_2 \vdash_v \mathsf{L}_1\ \diamond \quad \underset{loads}{\leftharpoonup} \mathsf{L}'$
- $\mathsf{L}_1\mathsf{L}_2 \vdash \mathsf{P}\,\diamond$
- P_1=$pr(\mathsf{P},\mathsf{L}_1)$

then

- $\vdash\ \mathsf{P},\mathsf{L}_1\mathsf{L}_2\mathsf{L}_3\ \diamond_a\ \Rightarrow\ \vdash\ \mathsf{P}\mathsf{P}_1,\mathsf{L}_2\mathsf{L}_3\ \diamond_a$
- $\mathsf{P},\mathsf{L}_1\mathsf{L}_2 \vdash \mathsf{t}\ \leq\ \mathsf{t}'\ \Rightarrow\ \mathsf{P}\mathsf{P}_1,\mathsf{L}_2\mathsf{L}' \vdash \mathsf{t}\ \leq\ \mathsf{t}'$
- $\mathsf{P},\mathsf{L}_1\mathsf{L}_2,\mathsf{E} \vdash \mathsf{e}\ :\ \mathsf{t}\ \Rightarrow\ \mathsf{P}\mathsf{P}_1,\mathsf{L}_2\mathsf{L}',\mathsf{E} \vdash \mathsf{e}\ :\ \mathsf{t}$
- $\mathsf{L}_2\mathsf{L}' \vdash \mathsf{P}\mathsf{P}_1\ \diamond$
- $\mathsf{P},\mathsf{E} \vdash_c \sigma\ \diamond\ \Rightarrow\ \mathsf{P}\mathsf{P}_1,\mathsf{E} \vdash_c \sigma\ \diamond$
- $\mathsf{P},\mathsf{L}_1\mathsf{L}_2,\mathsf{E} \vdash_r \sigma,\mathsf{e}\ :\ \mathsf{t}\ \Rightarrow\ \mathsf{P}\mathsf{P}',\mathsf{L}_2\mathsf{L}',\mathsf{E} \vdash_r \sigma,\mathsf{e}\ :\ \mathsf{t}$

[25] The assertion in the lemma is actually more general, because it also allows for further binaries L_0 to have been loaded, and P_0 to have been prepared.

4.4 Subject reduction and progress

Execution of a well-typed expression e does not overwrite objects, creates new objects in the free space, and does *not* affect the type of any expression e'' – even if e'' were a subexpression of e! Such a property is required for type soundness in imperative object oriented languages, and was proven, *e.g.* , in [5, 23]. In the current work this property holds only for well-typed expressions.

Lemma 4 *For* P, L, F, E, σ, *non-ground* e, t, *if*

- $\mathsf{L} \vdash \mathsf{P}\,\diamond$, *and*
- $\mathsf{P},\mathsf{L},\mathsf{E} \vdash_r \sigma,\mathsf{e}\ :\ \mathsf{t},$ *and*
- $\mathsf{e},\sigma,\mathsf{P},\mathsf{L}\ \leadsto\ \mathsf{e}',\sigma',\mathsf{P}',\mathsf{L}'$,

then

- $\sigma(\alpha)\ = \mathsf{c}\ \ \Rightarrow\ \ \sigma'(\alpha)\ = \mathsf{c},$
- $\sigma'(\alpha)\ = \mathsf{c}\ \ \Rightarrow\ \ \sigma(\alpha)\ = \mathsf{c}$ *or* α *free in* σ,
- $\mathsf{P},\mathsf{E} \vdash_c \sigma\ \diamond\ \ \Rightarrow\ \ \mathsf{P},\mathsf{E} \vdash_c \sigma'\ \diamond$
- $\mathsf{P},\mathsf{L},\mathsf{E} \vdash_r \sigma,\mathsf{e}''\ :\ \mathsf{t}''\ \ \Rightarrow\ \ \mathsf{P}',\mathsf{L}',\mathsf{E} \vdash_r \sigma',\mathsf{e}''\ :\ \mathsf{t}''$.

Proof by structural induction over the derivation $\leadsto$, and for the fourth part of the lemma, in the cases of VARASS or FLDACC1 by structural induction over the typing of e'', using the store conformance requirement whereby no object is stored within another object.

We can now prove progress and subject reduction:[26]

Lemma 5 *For any* P, L, F, E, σ, *non-ground* e, t, *if* $\mathsf{L} \vdash \mathsf{P}\,\diamond$, *and* $\mathsf{P},\mathsf{L},\mathsf{E} \vdash_r \sigma,\mathsf{e}\ :\ \mathsf{t}$
then there exist P', L', E', σ', e', t', *such that*
$\mathsf{e},\sigma,\mathsf{P},\mathsf{L}\ \leadsto\ \mathsf{e}',\sigma',\mathsf{P}',\mathsf{L}'$, *and*
$\mathsf{L}' \vdash \mathsf{P}'\,\diamond$, $\quad \vdash\ \mathsf{E}'\ \leq \mathsf{E}$, *and*

- $\mathsf{P}',\mathsf{L}',\mathsf{E}' \vdash_r \sigma',\mathsf{e}'\ :\ \mathsf{t}'$, *and* $\mathsf{P}',\mathsf{L}' \vdash \mathsf{t}'\ \leq\ \mathsf{t}$, *and* t=t' *if* e *is a non-l-ground variable,* *or*
- e' *contains the exception* $\mathsf{NllPErr}$, *or* $\mathsf{LoadErr}$, *or* $\mathsf{VerifErr}$, *or* $\mathsf{NoMethErr}$, *or* $\mathsf{NoFldErr}$, *or* $\mathsf{ClssChngErr}$.

Proof by structural induction over typing $\mathsf{P},\mathsf{L},\mathsf{E} \vdash_r \sigma,\mathsf{e}\ :\ \mathsf{t}$.

Thus, the new, possibly augmented, prepared code, P', preserves its well-formedness, and the store σ' preserves conformance. Uninitialized parts of the store, where $\sigma(\alpha)$ = $*$, are never de-referenced. Finally, execution never gets stuck.

[26] We assume an unlimited heap so that garbage collection is unnecessary.

Also, it is easy to prove that if execution of well-typed expressions e, loads some some types (*i.e.* if e rewrites according to LOAD or LOADERR), then e must have the form **new** c, or $\alpha[\alpha,t_2,t_3]^{\mathbf{i}}.m(\gamma)$. Namely, the well-typedness of the remaining type-contexts, *i.e.* field access $e_1[t_1,t_2].f$, and method call $e_1[t_1,t_2,t_3].m(e_2)$, requires the type of e_1 to be a subtype of t_1, which in its turn requires the presence of t_1 in P. Therefore, when executing verified code, the only expressions that may extend the loaded and prepared classes are object creation and interface method call.

5 Summary and alternatives

Verification of class c requires verification of all methods in c and all its (not yet prepared) superclasses. Verification of terms requires establishing subtype relations between types t and t′. If t has not been loaded yet, then it will be loaded with all its superclasses, except if t and t′ are identical, or t′ is an interface. Verification does not ensure the presence of fields or methods, it only ensures that all methods in a verified class respect their signatures. Resolution checks for the presence of fields and methods of given signatures. Thus the verifier relies on resolution to pick some of the possible errors, and resolution is safe only on code previously checked by the verifier.

Verification alone does not guard against link-time errors (*i.e.* LoadErr, or VerifErr, or NoMethErr, or NoFldErr, or ClssChngErr), but it does guarantee the integrity of the system. On the other hand, execution of unverified code may overwrite *any* part of the memory, and execute *any* methods.

Link-time errors can be created when running code that has been produced by a compiler, as shown in the various footnotes. However,link-time errors will not occur, one re-complies all importing classes/interfaces and all suclasses/subinterfaces after re-compiling a class or interface – we have not demonstrated this yet.

It is interesting that interfaces are treated by verification more leniently than classes, and thus require more run-time checks. It would have been possible to treat classes as leniently, or to treat interfaces more strictly.

In current implementations the boundary of decomposition are classes or interfaces. That is, we load several classes/interfaces together, and we verify several classes/interfaces together. Is it possible to consider other levels of decomposition? A probably less attractive, more lazy alternative would put the boundary of decomposition at methods, and would verify method bodies only before they are first called. This would make the judgment $L \vdash P\diamond$ even weaker, and would extend the operational semantics to check for previous verification

The integrity of the system is demonstrated by the subject reduction lemma. This is based on the well-typedness of the expression e and of the prepared code P. It is indicative, that in both judgments, namely in $L \vdash P\diamond$ and in $P, L, E \vdash_{\overline{r}} \sigma, e : t$ the role of the loaded code L is limited; the only information provided from L is which class/interface extends/implements which other class/interface, but the contents of the classes/interfaces in L is ignored.

A more lazy alternative, as suggested in in [9, 20] and formalized in [20], instead of immediately establishing that t is a subtype of t′ would post a constraint requiring a t to be a subtype of t′, to be validated only when t is loaded. This would treat L's as constraints, and the judgment $P, \epsilon, \vdash_{\overline{v}} e : t \;\; \leftharpoonup_{loads} L'$ to mean that the verifier established e to have type t, while posting L′.

It is easy to modify our model to express the above alternatives. More challenging would be a unified framework that would allow to characterize all such alternatives.

6 Conclusions, discussion and further work

We have given a model for the five components of execution, and have demonstrated how the corresponding checks together ensure type soundness. Our model describes these execution components at a high level, and distinguishes these components and the time of the associated checks. Thus, our account is useful for source language programmers, designers of new binary formats for Java, and designers of alternative distributions of the checks among the four components. Our model does not yet treat multiple loaders.

Formal treatments of linking were suggested in [3], albeit in a static setting. Dynamic linking at a fundamental level has been studied recently in [7, 1, 24], allowing for modules as first class values, usually untyped, concentrating on confluence and optimization issues. Recently, [15], discuss dynamic linking of native code as an extension of Typed Assembly Language.

Recent related work [20] complements ours, and provides a model of Java evaluation, dynamic preparation, verification and loading at the bytecode level, without interfaces, but with multiple loaders. Their approach is lazier than that of SUN implementations, and verification posts constraints as opposed to loading classes.

Further work includes refining the model to allow multiple class loaders (this would require the extension of the concept of class as *e.g.* [20]), extending the model to describe the source language and the compilation process, extending languages $\mathcal{L}$ and $\mathcal{P}$ with more Java features, considering different levels of decomposition, and applying the model to reconsider the meaning of binary compatibility [6].

Finally, though Java is novel in its approach to verification and dynamic linking, similar components and associated checks could be defined for any language that supports

A function $ld : \mathsf{Ident} \times \mathcal{P} \times \mathcal{L} \rightarrow \mathcal{L}$ is a *loader* iff:

$ld(\mathsf{t}, \mathsf{P}, \mathsf{L}) = \mathsf{L}' \quad \Rightarrow$

- $\mathsf{L}' \neq \epsilon \quad \Rightarrow \quad \mathsf{L}'(\mathsf{t}) \neq \epsilon$ and $\mathsf{P}(\mathsf{t}) = \mathsf{L}(\mathsf{t}) = \epsilon$
- $\vdash \mathsf{P}, \mathsf{L} \ \diamond_a \ \Rightarrow \ \vdash \mathsf{P}, \mathsf{LL}' \ \diamond_a$
- $\forall \mathsf{c}' : \quad \mathsf{L}'(\mathsf{c}') = \textbf{class}\ \mathsf{c}'\ \textbf{ext}\ \mathsf{c}''\ \textbf{impl}\ ...\mathsf{i}...\{...\} \ \Rightarrow\ \mathsf{PLL}'(\mathsf{c}'') \neq \epsilon,$ and $\mathsf{PLL}'(\mathsf{i}) \neq \epsilon$
- $\forall \mathsf{i} : \quad \mathsf{L}'(\mathsf{i}) = \textbf{interface}\ \mathsf{i}\ \textbf{ext}\ ...\mathsf{i}'...\{...\} \ \Rightarrow\ \mathsf{PLL}'(\mathsf{i}) \neq \epsilon$
- $\mathsf{L}' = \mathsf{L}'_1 \mathsf{L}'_2, \ \ \mathsf{L}'_1(\mathsf{c}) = \epsilon \ \Rightarrow\ ld(\mathsf{c}, \mathsf{P}, \mathsf{LL}'_1) = \mathsf{L}'_2$
- $\forall \mathsf{P}_0 : \ \vdash \mathsf{P}_0\mathsf{P}, \mathsf{LL}' \ \diamond_a \ \Rightarrow\ \mathsf{L}' = ld(\mathsf{c}, \mathsf{P}_0\mathsf{P}, \mathsf{L})$

A function $pr : \mathcal{P} \times \mathcal{L} \rightarrow \mathcal{P}$ is a *preparation* function iff:

$pr(\mathsf{P}, \mathsf{L}) = \mathsf{P}' \quad \Rightarrow$

- $\mathsf{P}'(\mathsf{c}) \neq \epsilon$ iff $\mathsf{L}(\mathsf{c}) \neq \epsilon$
- $\mathsf{P}'(\mathsf{i}) \neq \epsilon$ iff $\mathsf{L}(\mathsf{i}) \neq \epsilon$
- $\mathsf{P}'(\mathsf{c}) = \textbf{class}\ \mathsf{c}\ \textbf{ext}\ \mathsf{c}'\{\mathsf{t}_1\, \mathsf{f}_1\, \beta_1\ \dots\ \mathsf{t}_n\, \mathsf{f}_n\, \beta_n\ \ \mathsf{t}_{11}\, \mathsf{m}_1(\mathsf{t}_{12}\, \mathsf{x})\{\mathsf{e}_1\}\, \beta_{n+1},\ \dots\ \mathsf{t}_{q1}\, \mathsf{m}_1(\mathsf{t}_{q2}\, \mathsf{x})\{\mathsf{e}_q\}\, \beta_{n+q}\ \} \ \Rightarrow$
  - $\beta_i \neq \beta_j \qquad \forall i \neq j$ with $1 \leq i,j \leq n$, or $n+1 \leq i,j \leq n+q$
  - "c is defined in L, and the linked class ($\mathsf{P}'(\mathsf{c})$) has the same fields ($\mathsf{t}_1\, \mathsf{f}_1$ ... $\mathsf{t}_n\, \mathsf{f}_n$) as the original class ($\mathsf{L}(\mathsf{c})$)
    $\mathsf{L}(\mathsf{c}) = \textbf{class}\ \mathsf{c}\ \textbf{ext}\ \mathsf{c}'\{\mathsf{t}_1\, \mathsf{f}_1\ \dots\ \mathsf{t}_n\, \mathsf{f}_n\ \ \mathsf{t}'_{11}\, \mathsf{m}'_1(\mathsf{t}'_{12}\, \mathsf{x})\{\mathsf{e}'_1\},\ \dots\ \mathsf{t}'_{p1}\, \mathsf{m}'_1(\mathsf{t}'_{p2}\, \mathsf{x})\{\mathsf{e}'_p\}\ \}$
  - $\mathsf{c}' = \mathsf{Object}$ and $r=s=0$ or
    $\mathsf{PP}'(\mathsf{c}') = \textbf{class}\ \mathsf{c}'\ \textbf{ext}\ \mathsf{c}''\ \{\ \mathsf{t}''_1\, \mathsf{f}''_1\, \beta''_1\ ...\ \mathsf{t}''_r\, \mathsf{f}''_r\, \beta''_r\ \ \ \mathsf{t}''_{11}\, \mathsf{m}''_1(\mathsf{t}''_{12}\, \mathsf{x})\{\ \mathsf{e}''_1\ \}\, \beta''_{r+1}\ \ ...\ \ \mathsf{t}''_{s1}\, \mathsf{m}''_s(\mathsf{t}''_{s2}\, \mathsf{x})\{\ \mathsf{e}''_s\ \}\, \beta''_{r+s}\ \}$
  - "the methods in $\mathsf{P}'(\mathsf{c})$ are composed of ..."
    $\{\ \mathsf{t}_{11}\ \ \mathsf{m}_1(\mathsf{t}_{12}\ \mathsf{x})\{\mathsf{e}_1\ \}\ \beta_{n+1},\ \dots\ \mathsf{t}_{q1}\ \ \mathsf{m}_1(\mathsf{t}_{q2}\ \mathsf{x})\{\mathsf{e}_q\ \}\ \beta_{n+q}\ \ \} =$
    "... the methods inherited from c' , and not overridden in c"
    $\{\ \mathsf{t}''_{k1}\, \mathsf{m}''_k(\mathsf{t}''_{k2}\, \mathsf{x})\{\ \mathsf{e}''_k\ \}\, \beta''_{r+k}\ \ \mid\ \ 1 \leq k \leq s,$ and $\forall\, 1 \leq j \leq p:\ \mathsf{t}'_{j1}\, \mathsf{m}'_j(\mathsf{t}'_{j2}\, \mathsf{x}) \neq \mathsf{t}''_{k1}\, \mathsf{m}''_k(\mathsf{t}''_{k2}\, \mathsf{x})\ \ \}$
    ".... the methods from c which override a method from c', preserving the offset of the overridden method"
    $\cup\ \ \{\ \mathsf{t}''_{k1}\ \ \mathsf{m}''_k(\mathsf{t}''_{k2}\ \mathsf{x})\{\mathsf{e}\ \}\ \beta''_k\ \ \mid\ \ 1 \leq k \leq s,$ and $\exists\, 1 \leq j \leq p:\ \mathsf{t}'_{j1}\, \mathsf{m}'_j(\mathsf{t}'_{j2}\, \mathsf{x}) = \mathsf{t}''_{k1}\, \mathsf{m}''_k(\mathsf{t}''_{k2}\, \mathsf{x})$ and $\mathsf{e} = \mathsf{e}'_j\ \}$
    "... the methods newly introduced in c"
    $\cup\ \ \{\ \mathsf{t}'_{j1}\ \ \mathsf{m}'_j(\mathsf{t}'_{j2}\ \mathsf{x})\{\mathsf{e}_j\ \}\ \beta\ \ \mid\ \ 1 \leq j \leq p,$ and $\forall\ 1 \leq k \leq s:\ \mathsf{t}'_{j1}\, \mathsf{m}'_j(\mathsf{t}'_{j2}\, \mathsf{x}) \neq \mathsf{t}''_{k1}\, \mathsf{m}''_k(\mathsf{t}''_{k2}\, \mathsf{x})\ \ \}$
- $\mathsf{P}'(\mathsf{i}) = \textbf{interface}\ ...\ \Rightarrow\ \mathsf{P}'(\mathsf{i}) = \mathsf{L}(\mathsf{i})$

Note that the text enclosed in " and " is explanatory, and not part of the definition.

Figure 11. Loading and preparation

$$\mathcal{F}o(\mathsf{f},\mathsf{c},\mathsf{t},\mathsf{P}) = \begin{cases} \mathbf{-3} & \text{if } \mathsf{P}(\mathsf{c}) = \epsilon \\ \mathbf{-2} & \text{if } \mathsf{P}(\mathsf{c}) = \mathbf{interface} \ \ldots \\ \beta & \text{if } \mathsf{P}(\mathsf{c}) = \mathbf{class}\ \mathsf{c}\ \mathbf{ext}\ \mathsf{c}'\{\ \ldots\ \mathsf{t}\ \mathsf{f}\ \beta\ \ \ldots\} \\ \mathcal{F}o(\mathsf{f},\mathsf{c}',\mathsf{t},\mathsf{P}) & \text{if } \mathsf{P}(\mathsf{c}) = \mathbf{class}\ \mathsf{c}\ \mathbf{ext}\ \mathsf{c}'\{\ \mathsf{t}_1\ \mathsf{f}_1\ \beta_1 ... \mathsf{t}_\mathsf{n}\ \mathsf{f}_\mathsf{n}\ \beta_\mathsf{n}\ \mathsf{meths}\ \} \quad \text{and } \forall\, 1 \leq \mathsf{i} \leq \mathsf{n} :\ \mathsf{t}_\mathsf{i}\ \mathsf{f}_\mathsf{i} \neq \mathsf{t}\ \mathsf{f} \\ \mathbf{-1} & \text{if } \mathsf{c} = \mathsf{Object} \end{cases}$$

$$\mathcal{F}s(\mathsf{c},\mathsf{P}) = \begin{cases} \{\mathsf{t}_1\ \mathsf{f}_1\ \beta_1, \ldots\ \mathsf{t}_\mathsf{n}\ \mathsf{f}_\mathsf{n}\ \beta_\mathsf{n}\} \cup \mathcal{F}\mathsf{s}(\mathsf{c}',\mathsf{P}) & \text{if } \mathsf{P}(\mathsf{c}) = \mathbf{class}\ \mathsf{c}\ \mathbf{ext}\ \mathsf{c}'\{\mathsf{t}_1\ \mathsf{f}_1\ \beta_1, \ldots\ \mathsf{t}_\mathsf{n}\ \mathsf{f}_\mathsf{n}\ \beta_\mathsf{n}\ \mathit{meths}\} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathcal{M}o(\mathsf{m},\mathsf{c},\mathsf{t}_2,\mathsf{t}_1,\mathsf{P}) = \begin{cases} \mathbf{-3} & \text{if } \mathsf{P}(\mathsf{c}) = \epsilon \\ \mathbf{-2} & \text{if } \mathsf{P}(\mathsf{c}) = \mathbf{interface} \ ... \\ \beta & \text{if } \mathsf{P}(\mathsf{c}) = \mathbf{class}\ \mathsf{c}\ \mathbf{ext}\ \mathsf{c}'\{\ \ldots\ \mathsf{t}_1\ \mathsf{m}(\mathsf{t}_2\ \mathsf{x})\{\ \ldots\ \}\ \beta\ \ldots\} \\ \mathbf{-1} & \text{otherwise} \end{cases}$$

$$\mathcal{M}e(\beta,\mathsf{c},\mathsf{P}) = \begin{cases} \mathsf{e} & \text{if } \mathsf{P}(\mathsf{c}) = \mathbf{class}\ \mathsf{c}\ \mathbf{ext}\ \mathsf{c}'\{\ \ldots\ \mathsf{t}_1\ \mathsf{m}(\mathsf{t}_2\ \mathsf{x})\{\ \mathsf{e}\ \}\ \beta\ \ldots\} \\ \epsilon & \text{otherwise} \end{cases}$$

$$\mathcal{M}o^{\mathbf{i}}(\mathsf{m},\mathsf{i},\mathsf{t}_2,\mathsf{t}_1,\mathsf{P}) = \begin{cases} \mathbf{-3} & \text{if } \mathsf{P}(\mathsf{c}) = \epsilon \\ \mathbf{-2} & \text{if } \mathsf{P}(\mathsf{c}) = \mathbf{class} \ ... \\ \mathbf{0} & \text{if } \mathsf{P}(\mathsf{i}) = \mathbf{interface}\ \mathsf{i}\ \mathbf{ext}\ ...\{\ \ldots\ \mathsf{t}_1\ \mathsf{m}(\mathsf{t}_2\ \mathsf{x})\{\ \ldots\ \}\ \ldots\},\ \text{or } \mathcal{M}\mathsf{o}^{\mathbf{i}}(\mathsf{m},\mathsf{i}',\mathsf{t}_2,\mathsf{t}_1,\mathsf{P}) = \mathbf{0} \\ \mathbf{-1} & \text{otherwise} \end{cases}$$

Figure 12. The field and method look up functions $\mathcal{F}$, $\mathcal{M}$, $\mathcal{M}^{\mathbf{i}}$

some concept of modularity. The generalization of such ideas to other programming languages is an open issue.

Acknowledgments I am indebted to the TIC referees for extensible feedback and valuable suggestions. David Wragg gave invaluable initial information. Tatyana Valkevych, Susan Eisenbach, and Mark Skipper provided meticulous comments. Elena Zucca, and Eugenio Moggi asked pertinent questions and suggested a clearer presentation.

References

[1] Davide Ancona and Elena Zucca. A Primitive calculus for module systems. In *PPDP Proceedings*, September 1999.

[2] Martin Buechi. Type soundness Issues in Java, May 1999. Types mailing list, at http : //www.cis.upenn.edu/ bcpierce/types/archives and then /current/msg00140.html.

[3] Luca Cardelli. Program Fragments, Linking, and Modularization. In *POPL'97 Proceedings*, January 1997.

[4] Drew Dean. The Security of Static Typing with Dynamic Linking. In *Fourth ACM Conference on Computer and Communication Security*, 1997.

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

[6] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A Fragment Calculus - towards a model of Separate Compilation, Linking and Binary Compatibility. In *LICS Proceedings*, 1999.

[7] Kathleen Fisher, John Reppy, and Jon Riecke. A Calculus for Compiling and Linking Classes. In *ESOP Proceedings*, March 2000.

[8] Matthew Flatt, Shiram Khrishnamurthi, and Matthias Felleisen. Classes and Mixins. In *POPL Proceedings*, January 1998.

[9] Philip W. L. Fong and Robert D. Cameron. Proof linking: An architecture for modular verification of dynamically-linked mobile code. In *ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering (FSE'98)*, November 1998.

[10] Stephen N. Freund and J. C. Mitchell. A Formal Framework for the Java Bytecode Language and Verifier. In *OOPSLA Proceeedings*, November 1999.

[11] Stephen N. Freund and J. C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. In *OOPSLA Proceeedings*, October 1998.

[12] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996.

[13] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA Proceedings*, November 1999.

[14] Thomas Jensen, Daniel Le Metyayer, and Tommy Thorn. A Formalization of Visibility and Dynamic Loading in Java. In *IEEE ICCL*, 1998.

[15] Karl Krary, Michael Hicks, and Stephanie Weirich. Safe and Flexible Dynamic Linking of Native Code, May 2000. Internal Report, Univerity of Pennsylvania.

[16] Christopher League, Zhong Shao, and Valery Trifonov. Representing Java Classes in a Typed Intermediate language. In *ICFP Proceedings*, September 1999.

[17] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java^{TM} Virtual Machine. In *OOPSLA Proceedings*, October 1998.

[18] Tim Lindholm and Frank Yellin. *The Java Virtual Machine*. Addison-Wesley, 1997.

[19] Zhenyu Qian. Least Types for Memory Locations in Java Bytecode. In *FOOL 6*. http://www.cs.williams.edu/ kim/FOOL/sched6.html, 1999.

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

[21] Vijay Saraswat. Java is not type-safe. Technical report, AT&T Rresearch, 1997. http://www.research.att.comp/ vj/bug.html.

[22] Raymie Stata and Martin Abadi. A Type System For Java Bytecode Subroutines. In *POPL'98 Proceedings*, January 1998.

[23] Donald Syme. Proving Java Type Sound. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer, 1999.

[24] Joe Wells and Rene Vestergaard. Confluent Equational Reasoning for Linking with First-Class Primitive Modules. In *ESOP Proceedings*, March 2000.

[25] Phillip Yelland. Re: Type soundness Issues in Java, May 1999. Types mailing list, at http : //www.cis.upenn.edu/ bcpierce/types/archives and then /current/msg00145.html.

Another example demonstrating interfaces

The following example demonstrates the verifier's and run-time system's treatment of interfaces. It is an adaptation of the example which was posted by Martin Buechi [2] in the types mailing list, and was then discussed at some length.

We start with an interface Thinker implemented by class Man, and the class Main with method main:

```
interface Thinker { void be(); }

class Man impl Thinker {
   void be(){ System.out.println("be") ; }
}

class Main {
   public static void main (String args[] ) }
      Thinker descartes;
      Man john = new Man();
      System.out.println("a Man object created");
      if ( john instanceof Thinker)
         System.out.println("john is aThinker");
      else
         System.out.println("john is NOT a Thinker");
      descartes = new Man();
      System.out.println("a Man assigned to a Thinker") ;
      john.be();
}
```

We compile Thinker, Man and Main, and we then modify class Man, so that it does not implement Thinker, *i.e.*

```
class Man { void be(){ System.out.println("be") ; } }
```

We compile Man, without re-compiling Main. When we execute Main, we obtain the output:

```
a Man object created
john is NOT a Thinker
a Man assigned to a Thinker
IncompatibleClassChangeError :
     class Man does not implement Thinker
```

The above behavior is described by our model, namely:

- Verification of method main considers the assignment descartes = **new** Man(); as type correct, because the verifier is "liberal" with respect to interfaces
- Verification of the interface method call john.be() requires loading of the interface Thinker.
- Verification of method main does not need to load class Man.
- The assignment descartes = **new** Man(); is executed without any checks, and therefore without errors.
- The interface method call john.be() is compiled to a bytecode term which corresponds to john[Thinker,void,void]$^{\mathbf{i}}$.(). Execution of that term requires a run-time check according to rule INTFMETHCALL3. This check fails, and gives the error message IncompatibleClassChangeError : class Man does not implement Thinker