JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 5, Special Issue: Workshop on FTfJP, ECOOP 2006, June 2007

Modelling a JVM for polymorphic bytecode

Giovanni Lagorio, DISI, University of Genova, Italy

In standard compilation of Java-like languages, the bytecode generated for a given source depends on both the source itself and the compilation environment. This latter dependency poses some unnecessary restrictions on which execution environments can be used to run the code.
When using polymorphic bytecode, a binary depends only on its source and can be dynamically adapted to run on diverse environments.
Dynamic linking is particularly suited to polymorphic bytecode, because it can be adapted to an execution environment as late as possible, maximizing the flexibility of the approach.
We analyze how polymorphic bytecode can be dynamically linked presenting a deterministic model of a Java Virtual Machine which interleaves loading and linking steps with execution.
In our model, loading and execution phases are basically standard, whereas verification handles also type constraints, which are part of polymorphic bytecode, and resolution blends in verification.

1 INTRODUCTION

Java sources are compiled into `.class` binary files in order to be executed on a JVM (Java Virtual Machine). These binaries contain JVM instructions, better known as *bytecodes*, and other ancillary information.

Unsurprisingly, Java binaries contain many symbolic information: these are needed for linking classes compiled separately. Furthermore, Java supports dynamic linking so these information must be kept until runtime. Keeping references to class members in symbolic form inside binaries, as opposed to fixing object layouts at compile time[1], greatly enhances the possibility of reusing binaries in diverse binary environments.

For instance, when we compile the method invocation expression `Math.sin(0)`[2] the compiler generates a particular JVM instruction, `invokestatic`, along with some metadata containing the symbols `Math` and `sin`. This is legitimate: fixing a particular implementation of the class `Math` at compile time sounds premature.

It is perhaps less known that Java compilers, when generating binaries, are required to fix the types of the formal parameters and the return type of any invoked

[1] As it happens, for instance, in languages as C/C++.

[2] What this method does is immaterial in our example; however, as the reader probably knows, this standard method computes the sine of its argument.

Cite this article as follows: Giovanni Lagorio: *Modelling a JVM for polymorphic bytecode*, in Journal of Object Technology, vol. 6, no. 5, Special Issue: Workshop on FTfJP, ECOOP 2006, June 2007, pp. 29–59 http://www.jot.fm/issues/issues_2007_6/article2

method. In our example, a compiler would annotate the invocation of `Math.sin` with the type `double` for both the formal parameter and the return type. This dependency is not apparent from the source code and, we argue, unnecessary.

If the invocation `Math.sin(0)` appears in our source code inside, say, an invocation of `System.out.println`, then `sin` should be allowed to declare *any* return type[3]. Yet, this is not the case: Java compilation model poses some unnecessary restrictions on which execution environments can be used to run our code. For instance, using standard JVMs an implementation of class `Math` that uses `float`s instead of `double`s cannot be linked with our example even if its recompilation against such a math library would succeed.

Binaries would be more reusable if compilers did not enforce these invisible dependencies. Polymorphic bytecode [1], which has been proposed as a means for obtaining a compositional compilation for Java-like languages, makes (polymorphic) binaries dependent only on the sources they have been compiled from, employing *type variables* and accompanying *type constraints* stored inside binaries.

In [1] the focus is on compilation and the described linking process, necessary to *instantiate* polymorphic bytecode to standard monomorphic one, is static. However, as already noted [3, 4], combining polymorphic bytecode with dynamic linking allows programmers to reuse code with more flexibility, because the same polymorphic binaries can be dynamically adapted to run on diverse environments.

Of course, standard JVMs cannot directly execute polymorphic bytecode, as it contains type variables and type constraints. In a previous paper [11] we have analyzed how the JVM specification could be modified in order to make polymorphic bytecode run natively.

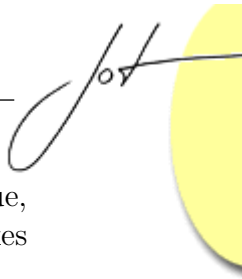
This paper improves our previous work. The main contribution of this paper is presenting a model closer to what an implementation would look like: now our runtime expressions resemble more closely actual JVM instructions. This is not a mere notation change: we have replaced the *instantiation phase*, where polymorphic bytecode was translated into standard monomorphic code (containing symbolic annotations), with a *JIT-compilation phase*[4].

The JIT-compilation directly transforms polymorphic bytecode into (an abstract version of) JVM instructions that do not contain any kind of symbolic annotations. The annotations contained in field accesses and method invocations are resolved once and for all into, respectively, field indexes and method indexes.

The former, field indexes, correspond to the positions of fields inside objects; an actual implementation would use an offset with respect to the beginning of objects in memory, while our model represents objects as tuples and uses a single slot for each field, no matter what the type of the field is.

The latter, method indexes, correspond to the positions of the methods inside

[3] Except for `void` that, technically [9], is not a type.
[4] JIT stands for "Just in time".



the virtual method tables; this is basically the standard implementation technique, though our virtual tables map indexes to method bodies instead of mapping indexes to *pointers* to the actual bodies.

Section 2 recalls how JVMs link and execute classes, and then discusses some design choices. Section 3 defines binary environments and describes the binary language we model; this section can be seen as a crash course in polymorphic bytecode and we refer to [1] for a complete presentation. Section 4 defines runtime expressions, Section 5 describes execution and presents some results. Section 6 discusses some implementation issues and, finally, Section 7 presents related work and concludes.

2 STANDARD JVM LINKING AND EXECUTION

Running a program, at the JVM level, actually means running a *class* `c`, that is, the `main` method of class `c`, in a certain *binary environment*. A binary environment is a collection of binaries, where the classes needed to execute `c` can be dynamically loaded[5] from.

Before a class can be executed, it must be loaded and linked. *Linking* consists of three different activities: *verification*, *preparation* and *resolution*. Verification ensures that binaries are structurally correct and that every instruction obeys the type discipline of the Java programming language [9]. If an error occurs during verification, then the exception `VerifyError` is thrown. Preparation, which we do not model, creates and initializes static fields. Resolution validates symbolic references to fields and methods[6]. If an error occurs during resolution, then the exception `IncompatibleClassChangeError` or one of its subclasses, for instance `NoSuchMethodError`, is thrown.

The JVM specification [12] does not impose an order of execution for loading and linking activities, as long as errors detected during linkage are thrown at a point in the execution where some action is taken by the program that might require linkage to the class or interface involved in the error. Standard JVMs are indeed quite lazy: they resolve symbolic references just before the execution of the instruction they are associated with.

In our model, loading and execution phases are basically standard, whereas verification handles also type constraints, which are part of polymorphic bytecode, and resolution blends in verification, because we chose to design the linking process as an incremental version of the inter-checking algorithm described in [1].

One drawback of this choice is that we need to resolve references earlier than standard JVMs; unfortunately, delaying the resolution of references gives rise to many issues when dealing with polymorphic bytecode.

The main advantage of our design is that it could be implemented on top of

[5] This is a simplified view: we are not considering class loaders [12, 13].

[6] Constructors are considered special methods, named `<init>`, at binary level.

$$
\begin{array}{rcl}
\mathcal{B} & ::= & \mathsf{b}_1 \ldots \mathsf{b}_n \\
\mathsf{b} & ::= & (\mathsf{cd}^{\mathsf{b}}, \bar{\gamma}) \\
 & & \\
\mathsf{cd}^{\mathsf{b}} & ::= & \texttt{class Object } \{\} \mid \texttt{class c extends c}' \texttt{ } \{ \overline{\mathsf{fd}^{\mathsf{b}}}\ \overline{\mathsf{md}^{\mathsf{b}}} \} \quad \textit{where } \mathsf{c} \neq \texttt{Object} \\
\overline{\mathsf{fd}^{\mathsf{b}}} & ::= & \mathsf{fd}^{\mathsf{b}}_1 \ldots \mathsf{fd}^{\mathsf{b}}_n \\
\mathsf{fd}^{\mathsf{b}} & ::= & \mathsf{c}\ \mathsf{f}; \\
\overline{\mathsf{md}^{\mathsf{b}}} & ::= & \mathsf{md}^{\mathsf{b}}_1 \ldots \mathsf{md}^{\mathsf{b}}_n \\
\mathsf{md}^{\mathsf{b}} & ::= & \mathsf{mh}\ \{\ \texttt{return}\ \mathsf{e}^{\mathsf{b}};\ \} \\
\mathsf{mh} & ::= & \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1, \ldots, \mathsf{c}_n\ \mathsf{x}_n) \\
\mathsf{e}^{\mathsf{b}} & ::= & \mathsf{x} \mid \mathsf{e}^{\mathsf{b}}[\mathsf{t.f}\ \mathsf{t}'] \mid \mathsf{e}^{\mathsf{b}}_0[\mathsf{t.m}(\bar{\mathsf{t}})\mathsf{t}'](\mathsf{e}^{\mathsf{b}}_1, \ldots, \mathsf{e}^{\mathsf{b}}_n) \mid \texttt{new}\ [\mathsf{c}\ \bar{\mathsf{t}}](\mathsf{e}^{\mathsf{b}}_1, \ldots, \mathsf{e}^{\mathsf{b}}_n) \mid \\
 & & (\mathsf{c})\mathsf{e}^{\mathsf{b}} \mid \ll \mathsf{c}, \mathsf{t} \gg \mathsf{e}^{\mathsf{b}} \\
\mathsf{t} & ::= & \mathsf{c} \mid \alpha \\
\bar{\mathsf{t}} & ::= & \mathsf{t}_1 \ldots \mathsf{t}_n \\
 & & \\
\gamma & ::= & \mathsf{t} \leq \mathsf{t}' \mid \phi(\mathsf{t}, \mathsf{f}, \mathsf{t}') \mid \mu(\mathsf{t}, \mathsf{m}, \bar{\mathsf{t}}, (\mathsf{t}', \bar{\mathsf{t}}')) \mid \kappa(\mathsf{c}, \bar{\mathsf{t}}, \bar{\mathsf{t}}') \mid \mathsf{c} \sim \mathsf{t} \\
\bar{\gamma} & ::= & \gamma_1 \ldots \gamma_n
\end{array}
$$

where class, field, method and parameter names in $\mathcal{B}$, $\overline{\mathsf{fd}^{\mathsf{b}}}$, $\overline{\mathsf{md}^{\mathsf{b}}}$ and mh are distinct

Figure 1: Binary environments.

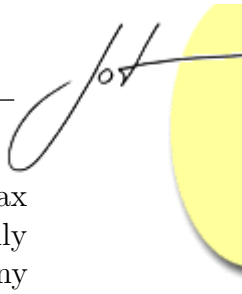
a standard JVM with a minimal effort: once a polymorphic fragment has been verified, it becomes a standard piece of code that can be handled like any other ".class". Delaying this transformation would instead impact on most components of a standard JVM, requiring a major revision. Section 6 discusses this topic in more depth.

3 BINARY ENVIRONMENTS

Binary environments are our abstraction of ".class" containers and are defined in Figure 1. Formally, a binary environment $\mathcal{B}$ is a sequence of binary fragments where each fragment defines a differently named class[7].

Binary fragments b are pairs consisting of a binary class declaration cd^{b} and a sequence of type constraints $\bar{\gamma}$. These constraints express the requirements that a binary environment $\mathcal{B}$ should meet in order to be compatible with cd^{b}. In other words, if $\bar{\gamma}$ hold in an environment $\mathcal{B}$, then cd^{b} can be run on $\mathcal{B}$ without getting stuck.

[7]This corresponds to the viewpoint of a JVM: when the binary of a certain class is searched, the first one found in the `CLASSPATH` is used, no matter how many other binaries may define the same class.



With the exception of some very small changes, we have inherited the syntax of binary class declarations and type constraints from [1]; the language is basically a binary version of Featherweight Java [10]. The superscript "b", used on many syntactic categories, means *binary*; for instance, a cd^{b} is a binary class declaration (that is, an abstract view of the bytecode contained in `.class` binary files). In [1] this superscript is used to distinguish between source and binary entities. Although we do not model any source level entity here, we keep the superscripts for two reasons: for consistency and for distinguishing between binary and *runtime* expressions, which we mark with the superscript "r".

Class declarations cd^{b} are either the declaration of the predefined class `Object` which, for simplicity, we assume declares no fields and methods, or the declaration of a class c, which contains a superclass name c', a sequence of field declarations $\overline{\mathsf{fd}^{\mathsf{b}}}$ and a sequence of method declarations $\overline{\mathsf{md}^{\mathsf{b}}}$.

Field and method declarations are standard, while binary expressions e^{b} deserve a detailed explanation. They are: parameter names, field accesses, method invocations, instance creations, casts and *polymorphic casts* (explained below).

Field accesses, method invocations and instance creations contain *annotations* between square brackets. These annotations reflect, in an abstract way, the actual encoding of those kinds of expression in Java bytecode.

Types $\mathtt{t}$ are either class names c (that is, the types ordinarily available at source level) or *type variables* α, which are instead inherent to the polymorphic approach and are not available to the source level programmer.

Let us describe annotations by means of an example: suppose we compile the source expression `anA.f.g`, where `anA` is a parameter of type `A`, in the following compilation environment:

```
class A {
    B f ;
}
class B {
    Object g ;
}
```

Because class `A` declares a field named `f` of type `B`, the subexpression `anA.f` is correct and has type `B`. Following the same reasoning, any standard Java compiler figures out that the whole expression is correct, has type `Object`, and generates the binary expression

$$\mathsf{e}^{\mathsf{b}}_{\mathtt{mono}} = \mathtt{anA[A.f\ B][B.g\ Object]}$$

The first annotation, `[A.f B]`, means that type `A` must provide (that is, inherit or declare) a field named `f` of type `B`. Analogously, class `B` must provide a field named `g` of type `Object`.

Method invocations and instance creations are annotated as well. The former

are annotated with the static type of the receiver, and the name, parameter type and return type of the method to be invoked. The latter are annotated with the class name and the parameter type of the constructor to be invoked.

Back to our example, the fact that field `f` must have exactly type `B` is deduced from the compilation environment, rather than explicitly expressed by the source code: while the programmer clearly wants a field named `g` from whatever `anA.f` is, there is no need for `anA.f` to have type `B` or `anA.f.g` to have type `Object`. Fixing all these types at compile time hinders the reusability of the code.

Indeed, the following environment

```
class A {
    C f ;
}
class C {
    Object g ;
}
```

obtained from the previous one by renaming class `B`, cannot be used to run $\mathsf{e}^{\mathsf{b}}_{\mathtt{mono}}$ even though the original source could be successfully recompiled in this environment.

Polymorphic bytecode solves this problem by fixing at compile time only the things that are known and cannot change. The code of our running example, for instance, would be compiled in the following polymorphic bytecode:

$$\mathsf{e}^{\mathsf{b}}_{\mathtt{poly}} = \mathtt{anA}[\mathtt{A.f}\ \alpha][\alpha.\mathsf{g}\ \beta]$$

where α and β are type variables. These variables can be replaced by class names when the execution environment, as opposed to the compilation environment, is known, making $\mathsf{e}^{\mathsf{b}}_{\mathtt{poly}}$ usable in more environments than $\mathsf{e}^{\mathsf{b}}_{\mathtt{mono}}$.

However, type variables are just a part of the solution. Of course, an arbitrary substitution of type variables into class names is not guaranteed to produce a sensible result. This is why we need type constraints too. The polymorphic binary expression $\mathsf{e}^{\mathsf{b}}_{\mathtt{poly}}$ should go hand in hand with the following type constraints:

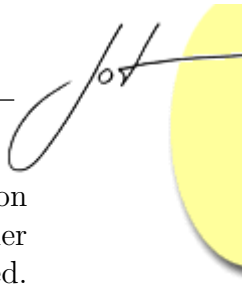
$$\bar{\gamma} = \phi(\mathtt{A}, \mathsf{f}, \alpha)\ \phi(\alpha, \mathsf{g}, \beta)$$

whose informal meaning is: "class `A` must provide a field named `f` of type α which, in turn, must provide a field named `g` of any type[8]". Indeed, we can find the value of α looking for a field named `f` in `A`; then, we either find the value of α (that is, the type `f` is declared of) or we know that no substitution can produce a sensible result[9].

The compilation of cast expressions presents another issue to take care of: consider the source expression e^{s} of type `t` and the expression: $\mathsf{e}^{\mathsf{s}}_{\mathtt{cast}} = (\mathsf{c})\mathsf{e}^{\mathsf{s}}$. This

[8] In this case we are assuming that the variable β is not used in any constraint and can assume any value.

[9] If `A` is unavailable or does not provide a field `f`, then no substitution can satisfy $\phi(\mathtt{A}, \mathsf{f}, \alpha)$.



cast is correct whenever $\mathtt{t}$ and $\mathtt{c}$ are in subtype relation, however the translation of an upcast is different from the translation of a downcast. Indeed, in the former case the cast is just discarded, while in the latter case a runtime check is required. If the relation between $\mathtt{t}$ and $\mathtt{c}$ is unknown, then the polymorphic cast expression $\ll \mathtt{c}, \mathtt{t} \gg \mathtt{e}^{\mathtt{b}}$ can be used. When polymorphic bytecode is *instantiated*, that expression is replaced by $\mathtt{e}^{\mathtt{b}}$ in binary environments where $\mathtt{t}$ is more specific than $\mathtt{c}$, and by a standard cast $(\mathtt{c})\mathtt{e}^{\mathtt{b}}$ in the others.

The bottom of Figure 1 shows the five kinds of constraints that we need; their informal meaning is the following:

- $\mathtt{t} \leq \mathtt{t}'$ — type $\mathtt{t}$ is a subtype of $\mathtt{t}'$
- $\phi(\mathtt{t}, \mathtt{f}, \mathtt{t}')$ — type $\mathtt{t}$ provides a field named $\mathtt{f}$ of type $\mathtt{t}'$
- $\mu(\mathtt{t}, \mathtt{m}, \bar{\mathtt{t}}, (\mathtt{t}', \bar{\mathtt{t}}'))$ — type $\mathtt{t}$ provides a method named $\mathtt{m}$, applicable to argument types $\bar{\mathtt{t}}$, with parameter types $\bar{\mathtt{t}}'$ and return type $\mathtt{t}'$ (the subtle reason why this kind of constraint and the following one need to consider both the formal and the actual parameter types is explained below)
- $\kappa(\mathtt{c}, \bar{\mathtt{t}}, \bar{\mathtt{t}}')$ — class $\mathtt{c}$ provides a constructor applicable to argument types $\bar{\mathtt{t}}$, with parameter types $\bar{\mathtt{t}}'$
- $\mathtt{c} \sim \mathtt{t}$ — class $\mathtt{c}$ and type $\mathtt{t}$ are comparable.

These are the constraints given in [1], with the exception of constraints "$\exists\, \mathtt{c}$", with the informal meaning "class $\mathtt{c}$ must exist". Indeed, these existential constraints are only needed in the static approach to make compositional compilation equivalent to standard global compilation. In a JVM we do not need to require the existence of all classes named in the sources: if a class is not needed for the execution, then we do not care whether such a class exists.

As said, method and constructor resolution constraints need to deal with both the formal parameter types and the actual parameter types. In [1], these are needed to obtain standard bytecode, where invocations need to be annotated with formal parameter types. By forbidding overloading (which we have decided not to, see below) we could simplify the constraints; anyway, keeping the distinction between formal parameter types and actual parameter types makes method lookups simpler. Consider, for instance, the following example:

```
class C {}
class D extends C {}
class Test {
    Test m(C aC) { return this ; }
}
...
```

$$
\begin{array}{rcl}
\mathsf{e}^{\mathsf{r}} & ::= & v \mid \mathsf{e}^{\mathsf{r}}.\texttt{getfield}(n) \mid \mathsf{e}^{\mathsf{r}}.\texttt{invokevirtual}(n, \overline{\mathsf{e}^{\mathsf{r}}}) \mid \\
 & & \mathsf{c}.\texttt{new}(\overline{\mathsf{e}^{\mathsf{r}}}) \mid \texttt{aload}(n) \mid \mathsf{e}^{\mathsf{r}}.\texttt{checkcast}(\mathsf{c}) \mid \\
 & & \epsilon \mid \texttt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}) \mid \texttt{bootstrap}(\bar{\gamma}, \mathsf{e}^{\mathsf{b}}) \\
\overline{\mathsf{e}^{\mathsf{r}}} & ::= & \mathsf{e}^{\mathsf{r}}_1, \ldots, \mathsf{e}^{\mathsf{r}}_n \\
v & ::= & \texttt{new}\ \mathsf{c}(\bar{v}) \\
\bar{v} & ::= & v_1, \ldots, v_n \\
\epsilon & ::= & \texttt{NoClassDefFoundError} \mid \texttt{ClassCircularityError} \mid \texttt{VerifyError} \mid \\
 & & \texttt{ClassCastException} \\
n & ::= & 0 \mid 1 \mid 2 \mid \ldots
\end{array}
$$

Figure 2: Syntax of runtime expressions.

```
new Test().m(new C()) ; // μ(Test, m, C, α)
new Test().m(new D()) ; // μ(Test, m, D, β)
```

If method invocation constraints did not contain type variables representing the formal paramter types, as in the comments of the above example, knowing that both constraints would hold using the substitution $\{\alpha \mapsto \texttt{Test}, \beta \mapsto \texttt{Test}\}$ would not give us any information on which method is to be invoked unless, as we mentioned, we forbid overloading; in that case, there could be at most a method named `m` declared in `Test` or its superclasses.

4 RUNTIME EXPRESSIONS

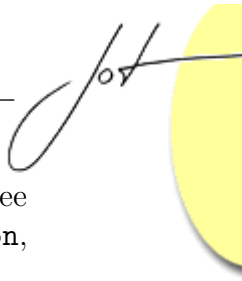
Figure 2 shows runtime expressions; except for `verifyCls` and `bootstrap`, which are peculiar to our approach and are described, respectively, below and in the next section, they are: values v, field accesses, method invocations, instance creations, `this`/parameter fetching, cast expressions and exceptions ϵ. As said, their syntax recalls the syntax of actual JVM instructions.

Note that `aload` instructions are contained only in the body of the methods and are never directly executed. That is, these instructions are replaced by the target object and the actual parameters when a method invocation is executed (see the second metarule in Figure 12).

Values v represent objects; each object consists of the keyword `new`, followed by its class name and the sequence of its field values between round brackets.

Field accesses and method invocations are annotated by an integer index. This index indicates the position of the field to be fetched for field accesses, and the position of the body to be executed for method invocations.

Exceptions ϵ are: `NoClassDefFoundError`, thrown when a needed class cannot be found, `ClassCircularityError`, thrown when loading a certain class would introduce a cycle in the inheritance hierarchy, `VerifyError`, thrown when the checking



of a type constraint fails or when type constraints are not strong enough to guarantee the safe execution of the class they are associated with, and `ClassCastException`, thrown when the execution of a cast fails.

The special expression `verifyCls` is used to wrap an expression e^{r} when the execution of e^{r} is stuck because it needs some class `c` to be verified.

The only expressions that can trigger this behaviour in our model are instance creations: the creation of an object of type `c` can happen only if class `c` has been successfully verified (this action, in turn, may require other classes to be loaded).

So, in an environment where `c` has not been verified yet, the expression $\mathsf{e}^{\mathsf{r}}_1 = \texttt{new c}(\ldots)$ is rewritten into $\mathsf{e}^{\mathsf{r}}_2 = \texttt{verifyCls}(\texttt{c}, \mathsf{e}^{\mathsf{r}}_1)$, that can be read as "verify class `c` first, and then go on with the execution of $\mathsf{e}^{\mathsf{r}}_1$".

Rewrite rules (Section 5) guarantee that either the execution of $\mathsf{e}^{\mathsf{r}}_1$ will restart in a new environment where class `c` has been successfully verified, *or* the whole expression $\mathsf{e}^{\mathsf{r}}_2$ will be rewritten into a loading/verification exception.

5 EXECUTION

Execution, modeled in a small step style, has the form: $\mathcal{B}^{\mathsf{L}}_1, \mathcal{M}_1, \mathsf{e}^{\mathsf{r}}_1 \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \mathcal{M}_2, \mathsf{e}^{\mathsf{r}}_2$ where:

- $\mathcal{B}$ is the execution environment where classes are loaded from; it contains polymorphic binary fragments.
- $\mathcal{B}^{\mathsf{L}}_1$ and $\mathcal{B}^{\mathsf{L}}_2$ contain the *loaded* classes; they are contained[10] in $\mathcal{B}$, and $\mathcal{B}^{\mathsf{L}}_2$ is always equal to or greater than $\mathcal{B}^{\mathsf{L}}_1$.
- $\mathcal{M}_1$ and $\mathcal{M}_2$ are the environments of *verified* classes. These environments map class names to virtual method tables ν. Virtual method tables ν, in turn, map integer indexes to the runtime expressions that correspond to method bodies.
- $\mathsf{e}^{\mathsf{r}}_1$ and $\mathsf{e}^{\mathsf{r}}_2$ are the expressions to execute.

No rewrite rule changes all three components at once: the rewriting rules for loading classes act only on $\mathcal{B}^{\mathsf{L}}$, the ones for linking on $\mathcal{M}$, and the ones for standard execution on e^{r}.

Execution starts with the special expression `bootstrap` from the empty environments of loaded and verified classes, that is either rewritten into a runtime expression, when verification succeeds, or into an exception, if constraints $\bar{\gamma}$ are not strong enough to guarantee a safe execution for e^{b} or if verification of $\bar{\gamma}$ fails (more details on `bootstrap` are given below).

[10] Formally they are sequences, but we assume that no binary environment contains different declarations for the same class, so we can consider them as maps when this simplifies the discussion.

Constraint verification is modeled by the execution of *verification actions* $\mathcal{V}$. The execution of these actions can either produce a new action, to go on with the verification, or produce a final result: a substitution σ, when the verification succeeds, or an exception ϵ, when the verification fails. Substitutions produced by successful verifications map the type variables contained in the constraints to actual type names (of the current environment) that satisfy the constraints.

Because the verification of a class can never trigger the verification of another class, the execution of verification actions does not need to know or update the set of verified classes. So, verification has the form $\mathcal{B}_1^{\mathsf{L}}, \mathcal{V}_1 \rightarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{V}_2$, where:

$$\begin{array}{rcl} \mathcal{V} & ::= & \mathtt{load}(\mathtt{c}, \mathcal{V}) \mid \mathtt{verify}(\bar{\gamma}, \mathcal{V}) \mid \mathtt{verifyEither}(\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3) \mid \\ & & \mathtt{match}(\bar{\mathtt{t}}, \bar{\mathtt{c}}, \mathcal{V}) \mid \sigma \mid \epsilon \end{array}$$

The informal meaning of actions $\mathcal{V}$ is, respectively,

- load $\mathtt{c}$, and then execute $\mathcal{V}$;
- verify $\bar{\gamma}$, and then execute $\mathcal{V}$;
- verify either $\mathcal{V}_1$ or $\mathcal{V}_2$, and then execute $\mathcal{V}_3$;
- produce a substitution σ matching $\bar{\mathtt{t}}$ with $\bar{\mathtt{c}}$, and then execute $\sigma(\mathcal{V})$ – note that this is the standard application of a substitution except when σ is applied to another σ' (inside $\mathcal{V}$): in this case the result of the substitution is the composition of σ and σ';
- the verification has succeeded and the result is the substitution σ,
- the verification has failed and the exception ϵ has to be thrown.

In the following subsections we first describe the loading process and class verification (which uses JIT-compilation and constraint verification that are detailed in their own subsection). Then, we show how the system can be bootstrapped and, finally, we prove some useful properties.

Loading

The rewrite rules for action `load` are shown in Figure 3: If the requested class cannot be found or its loading would introduce a cycle in the type hierarchy, then the corresponding exception is thrown (first and second rules). If everything is fine then the verification continues with $\mathcal{V}$ in a new environment $\mathcal{B}_2^{\mathsf{L}}$ where the binary $\mathtt{b}$, loaded from $\mathcal{B}$, has been added to $\mathcal{B}_1^{\mathsf{L}}$. Note that we check that loading a class does not create cycles in the type hierarchy, whereas we do not check overriding rules; in this regard we do exactly what standard JVMs do.

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \texttt{load}(\mathsf{c}, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \texttt{NoClassDefFoundError}} \quad \mathsf{c} \notin def(\mathcal{B})$$

$$\frac{}{\mathcal{B}_1^{\mathsf{L}}, \texttt{load}(\mathsf{c}, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}_1^{\mathsf{L}}, \texttt{ClassCircularityError}} \quad \begin{array}{l} \mathsf{c} \in def(\mathcal{B} \setminus \mathcal{B}_1^{\mathsf{L}}) \\ \mathcal{B}_2^{\mathsf{L}} = \mathcal{B}_1^{\mathsf{L}}\, \mathcal{B}(\mathsf{c}) \\ isInsideACycle_{\mathcal{B}_2^{\mathsf{L}}}(\mathsf{c}) \end{array}$$

$$\frac{}{\mathcal{B}_1^{\mathsf{L}}, \texttt{load}(\mathsf{c}, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{V}} \quad \begin{array}{l} \mathsf{c} \in def(\mathcal{B} \setminus \mathcal{B}_1^{\mathsf{L}}) \\ \mathcal{B}_2^{\mathsf{L}} = \mathcal{B}_1^{\mathsf{L}}\, \mathcal{B}(\mathsf{c}) \\ \neg isInsideACycle_{\mathcal{B}_2^{\mathsf{L}}}(\mathsf{c}) \end{array}$$

$$isInsideACycle_{\mathcal{B}}(\mathsf{c}_0) = \exists n \geq 0 : \ super_{\mathcal{B}}(\mathsf{c}_0) = \mathsf{c}_1 \ \wedge \ \ldots \ \wedge \ super_{\mathcal{B}}(\mathsf{c}_n) = \mathsf{c}_0$$
$$super_{\mathcal{B}}(\mathsf{c}) = super(classDeclaration(\mathcal{B}(\mathsf{c})))$$
$$classDeclaration(\ (\mathsf{cd}^{\mathsf{b}}, \bar{\gamma})\ ) = \mathsf{cd}^{\mathsf{b}}$$
$$super(\texttt{class c extends c}' \ \{\ \overline{\mathsf{fd}^{\mathsf{b}}}\ \overline{\mathsf{md}^{\mathsf{b}}}\ \}) = \mathsf{c}'$$

Figure 3: Rewrite rules for loading classes.

In [1] method overloading and field hiding are not modeled, so we resolve type constraints without taking these two features into account[11]. However, we do not need to forbid the presence of overloaded methods or hidden fields, that is, we do not check their presence when we load a class.

In checking constraints, when we search for a method named m, invoked with n arguments, we end the lookup procedure at the first m accepting n arguments. If overloading were supported, we should collect all the applicable methods and find the most specific for the invocation, exactly as a standard Java *compiler* does.

Analogously, when we search for a field f we end the lookup procedure at the first field named f.

Class verification

As said in the previous section, class verification, and the subsequent introduction of successfully verified classes in the system, is carried out by the execution of the special expression `verifyCls`, whose rewrite rules are shown in Figure 4. The first four rules encode, respectively, that: there is no need to verify a class twice, to verify a class we must load it first (second and third rules), and classes are verified after their superclass.

The next three rules are more interesting and use two auxiliary predicates defined

[11] Type constraints would probably have to be changed to model overloading and hiding fully.

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathcal{M}, \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{M}, \mathsf{e}^{\mathsf{r}}} \quad \mathsf{c} \in \mathit{def}(\mathcal{M})$$

$$\frac{\mathcal{B}_1^{\mathsf{L}}, \mathtt{load}(\mathsf{c}, \emptyset) \rightarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \emptyset}{\mathcal{B}_1^{\mathsf{L}}, \mathcal{M}, \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{M}, \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}})} \quad \mathsf{c} \notin \mathit{def}(\mathcal{B}_1^{\mathsf{L}})$$

$$\frac{\mathcal{B}^{\mathsf{L}}, \mathtt{load}(\mathsf{c}, \emptyset) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \epsilon}{\mathcal{B}^{\mathsf{L}}, \mathcal{M}, \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{M}, \epsilon} \quad \mathsf{c} \notin \mathit{def}(\mathcal{B}^{\mathsf{L}})$$

$$\frac{}{\begin{array}{l}\mathcal{B}^{\mathsf{L}}, \mathcal{M}, \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}) \rightsquigarrow_{\mathcal{B}} \\ \mathcal{B}^{\mathsf{L}}, \mathcal{M}, \mathtt{verifyCls}(\mathsf{c}', \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}))\end{array}} \quad \begin{array}{l}\mathsf{c} \in \mathit{def}(\mathcal{B}^{\mathsf{L}}) \setminus \{\mathtt{Object}\} \\ \mathsf{c}' = \mathit{super}_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}) \\ \mathsf{c}' \notin \mathit{def}(\mathcal{M})\end{array}$$

$$\frac{\begin{array}{c}\mathcal{B}_1^{\mathsf{L}}, \mathtt{verify}(\bar{\gamma}, \emptyset) \rightarrow_{\mathcal{B}}^{+} \mathcal{B}_2^{\mathsf{L}}, \sigma \\ \mathcal{B}_2^{\mathsf{L}}, \mathcal{M}_1, \sigma \vdash \mathsf{c} \Rightarrow \mathcal{M}_2\end{array}}{\mathcal{B}_1^{\mathsf{L}}, \mathcal{M}_1, \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{M}_2, \mathsf{e}^{\mathsf{r}}} \quad \begin{array}{l}\mathtt{readyTBV}(\mathsf{c}, \mathcal{B}_1^{\mathsf{L}}, \mathcal{M}_1) \\ (\mathsf{cd}^{\mathsf{b}}, \bar{\gamma}) = \mathcal{B}_1^{\mathsf{L}}(\mathsf{c}) \\ \mathtt{wellFormedAndCompliant}(\bar{\gamma}, \mathsf{cd}^{\mathsf{b}})\end{array}$$

$$\frac{\mathcal{B}_1^{\mathsf{L}}, \mathtt{verify}(\bar{\gamma}, \emptyset) \rightarrow_{\mathcal{B}}^{+} \mathcal{B}_2^{\mathsf{L}}, \epsilon}{\mathcal{B}_1^{\mathsf{L}}, \mathcal{M}_1, \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{M}_1, \epsilon} \quad \begin{array}{l}\mathtt{readyTBV}(\mathsf{c}, \mathcal{B}_1^{\mathsf{L}}, \mathcal{M}_1) \\ (\mathsf{cd}^{\mathsf{b}}, \bar{\gamma}) = \mathcal{B}_1^{\mathsf{L}}(\mathsf{c}) \\ \mathtt{wellFormedAndCompliant}(\bar{\gamma}, \mathsf{cd}^{\mathsf{b}})\end{array}$$

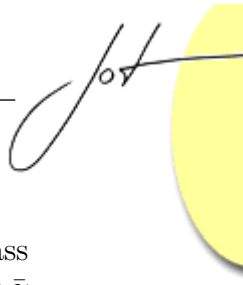
$$\frac{}{\begin{array}{l}\mathcal{B}^{\mathsf{L}}, \mathcal{M}, \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}) \rightsquigarrow_{\mathcal{B}} \\ \mathcal{B}^{\mathsf{L}}, \mathcal{M}, \mathtt{VerifyError}\end{array}} \quad \begin{array}{l}\mathtt{readyTBV}(\mathsf{c}, \mathcal{B}^{\mathsf{L}}, \mathcal{M}) \\ (\mathsf{cd}^{\mathsf{b}}, \bar{\gamma}) = \mathcal{B}^{\mathsf{L}}(\mathsf{c}) \\ \neg\mathtt{wellFormedAndCompliant}(\bar{\gamma}, \mathsf{cd}^{\mathsf{b}})\end{array}$$

$$\mathtt{readyTBV}(\mathsf{c}, \mathcal{B}^{\mathsf{L}}, \mathcal{M}) = (\mathsf{c} \in \mathit{def}(\mathcal{B}^{\mathsf{L}}) \setminus \mathit{def}(\mathcal{M})) \wedge (\mathsf{c} = \mathtt{Object} \vee \mathit{super}_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}) \in \mathit{def}(\mathcal{M}))$$
$$\mathtt{wellFormedAndCompliant}(\bar{\gamma}, \mathsf{cd}^{\mathsf{b}}) = \mathit{wellFormed}(\bar{\gamma}) \wedge \bar{\gamma} \vdash \mathsf{cd}^{\mathsf{b}}\diamond$$

Figure 4: Rewrite rules for verifying and linking classes.

at the bottom of the figure: `readyTBV` and `wellFormedAndCompliant`. The former predicate expresses that a class is "ready To Be Verified" when the class has been loaded, has not been verified yet and its superclass, if any, has already been verified. The latter encodes the requirements on the constraints $\bar{\gamma}$ accompanying a binary class cd^{b}: they must be well-formed and cd^{b} must be *compliant* with them. The notation $\rightarrow^{+}$ indicates the standard transitive closure of the relation $\rightarrow$.

Well-formedness of sequences of type constraints is defined in [1] and guarantees that well-formed sequences can be reordered in a way that allows the checking of them (w.r.t. a type environment) with a single iteration. At each step of such an iteration a constraint γ is processed, finding either a substitution which makes γ



hold in the current environment or a proof that no substitution exists.

The judgment $\bar{\gamma} \vdash \mathsf{cd}^{\mathsf{b}}\diamond$, shown in Figure 5 and described below, to be read "class declaration cd^{b} is compliant with type constraints $\bar{\gamma}$", holds when type constraints $\bar{\gamma}$ are strong enough to guarantee the safe execution of cd^{b}. Following the terminology introduced in [5], this corresponds to the *intra-checking* of cd^{b}, while *inter-checking* happens incrementally when executing `match` actions (triggered by verification).

Back to the fifth rule: if all the above conditions are met and $\bar{\gamma}$ are successfully verified, starting from the empty substitution $\emptyset$ and producing the substitution σ (premise of the rule), then the execution of e^{r} continues in $\mathcal{B}_2^{\mathsf{L}}$ (verification may require to load new classes) and $\mathcal{M}_2$. The environment $\mathcal{M}_2$ is obtained by the JIT-compilation of class c, corresponding to the judgment

$$\mathcal{B}_2^{\mathsf{L}}, \mathcal{M}_1, \sigma \vdash \mathsf{c} \Rightarrow \mathcal{M}_2$$

to be read "given the binary environment of loaded classes $\mathcal{B}_2^{\mathsf{L}}$, the environment

$$\frac{}{\bar{\gamma} \vdash \texttt{class Object } \{\}\diamond} \qquad \frac{\bar{\gamma}, \mathsf{c} \vdash \overline{\mathsf{md}^{\mathsf{b}}}\diamond}{\bar{\gamma} \vdash \texttt{class c extends c}' \ \{\ \overline{\mathsf{fd}^{\mathsf{b}}}\ \overline{\mathsf{md}^{\mathsf{b}}}\ \}\diamond}$$

$$\frac{i \in \{1,\ldots,n\}\ \bar{\gamma}, \mathsf{c} \vdash \mathsf{md}_i^{\mathsf{b}}\diamond}{\bar{\gamma}, \mathsf{c} \vdash \mathsf{md}_1^{\mathsf{b}} \ldots \mathsf{md}_n^{\mathsf{b}}\diamond} \quad n > 1 \qquad \frac{\begin{array}{l}\bar{\gamma}, \{\texttt{this} \mapsto \mathsf{c}, \mathsf{x}_1 \mapsto \mathsf{c}_1, \ldots, \mathsf{x}_n \mapsto \mathsf{c}_n\} \vdash \mathsf{e}^{\mathsf{b}} : \mathsf{t} \\ \bar{\gamma} \vdash \mathsf{t} \leq \mathsf{c}_0\end{array}}{\bar{\gamma}, \mathsf{c} \vdash \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1, \ \ldots, \ \mathsf{c}_n\ \mathsf{x}_n)\ \{\ \texttt{return } \mathsf{e}^{\mathsf{b}}\texttt{;}\ \}\diamond}$$

$$\frac{}{\bar{\gamma}, \Pi \vdash \mathsf{x} : \mathsf{c}} \quad \Pi(\mathsf{x}) = \mathsf{c} \qquad \frac{\begin{array}{l}\bar{\gamma}, \Pi \vdash \mathsf{e}^{\mathsf{b}} : \mathsf{t} \\ \bar{\gamma} \vdash \phi(\mathsf{t}, \mathsf{f}, \mathsf{t}')\end{array}}{\bar{\gamma}, \Pi \vdash \mathsf{e}^{\mathsf{b}}[\mathsf{t}.\mathsf{f}\ \mathsf{t}'] : \mathsf{t}'}$$

$$\frac{\begin{array}{l}i \in \{0,\ldots,n\}\ \bar{\gamma}, \Pi \vdash \mathsf{e}_i^{\mathsf{b}} : \mathsf{t}_i \\ \bar{\gamma} \vdash \mu(\mathsf{t}_0, \mathsf{m}, \mathsf{t}_1 \ldots \mathsf{t}_n, (\mathsf{t}', \bar{\mathsf{t}}'))\end{array}}{\bar{\gamma}, \Pi \vdash \mathsf{e}_0^{\mathsf{b}}[\mathsf{t}_0.\mathsf{m}(\bar{\mathsf{t}}')\mathsf{t}'](\mathsf{e}_1^{\mathsf{b}}, \ \ldots, \ \mathsf{e}_n^{\mathsf{b}}) : \mathsf{t}'} \qquad \frac{\begin{array}{l}i \in \{1,\ldots,n\}\ \bar{\gamma}, \Pi \vdash \mathsf{e}_i^{\mathsf{b}} : \mathsf{t}_i \\ \bar{\gamma} \vdash \kappa(\mathsf{c}, \mathsf{t}_1 \ldots \mathsf{t}_n, \bar{\mathsf{t}}')\end{array}}{\bar{\gamma}, \Pi \vdash \texttt{new } [\mathsf{c}\ \bar{\mathsf{t}}'](\mathsf{e}_1^{\mathsf{b}}, \ldots, \ \mathsf{e}_n^{\mathsf{b}}) : \mathsf{c}}$$

$$\frac{\begin{array}{l}\bar{\gamma}, \Pi \vdash \mathsf{e}^{\mathsf{b}} : \mathsf{c}' \\ \bar{\gamma} \vdash \mathsf{c} \sim \mathsf{c}'\end{array}}{\bar{\gamma}, \Pi \vdash (\mathsf{c})\mathsf{e}^{\mathsf{b}} : \mathsf{c}} \qquad \frac{\begin{array}{l}\bar{\gamma}, \Pi \vdash \mathsf{e}^{\mathsf{b}} : \mathsf{t} \\ \bar{\gamma} \vdash \mathsf{c} \sim \mathsf{t}\end{array}}{\bar{\gamma}, \Pi \vdash \ll \mathsf{c}, \mathsf{t} \gg \mathsf{e}^{\mathsf{b}} : \mathsf{c}}$$

$$\frac{\bar{\gamma} \vdash \mathsf{t}_0 \leq \mathsf{t}_1}{\bar{\gamma} \vdash \mathsf{t}_0 \sim \mathsf{t}_1} \qquad \frac{\bar{\gamma} \vdash \mathsf{t}_1 \leq \mathsf{t}_0}{\bar{\gamma} \vdash \mathsf{t}_0 \sim \mathsf{t}_1}$$

$$\frac{\bar{\gamma} \vdash \mathsf{t}_0 \leq \mathsf{t}_1\ \bar{\gamma} \vdash \mathsf{t}_1 \leq \mathsf{t}_2}{\bar{\gamma} \vdash \mathsf{t}_0 \leq \mathsf{t}_2} \qquad \frac{}{\bar{\gamma} \vdash \gamma_i} \quad \bar{\gamma} = \gamma_1 \ldots \gamma_n$$

Figure 5: Compliance of code and constraints.

of verified classes $\mathcal{M}_1$ and the substitution σ, JIT-compiling class $\mathtt{c}$ produces the verified class environment $\mathcal{M}_2$". The definition of this judgment is explained below.

The remaining two rules of the figure deal with error cases.

Metarules for compliance, defined in Figure 5, encode that: declaration of class `Object` is compliant with any set of constraints since it has no requirements, declaration of class $\mathtt{c}$ is compliant with $\bar{\gamma}$ if all its methods are, and a method declaration is compliant with $\bar{\gamma}$ when its body can be typechecked and has a type that is a subtype of the declared return type in $\bar{\gamma}$.

Compliance of expressions is modeled by the judgment $\bar{\gamma}, \Pi \vdash \mathtt{e^b} : \mathtt{t}$ to be read "binary expression $\mathtt{e^b}$, in the local environment Π, is compliant with type constraints $\bar{\gamma}$ and has type $\mathtt{t}$". Local environment Π maps parameter names to their declared type and the implicit parameter `this` to the current class name.

Note that, in Figure 5, while there are some trivial closures for subtyping, the basic subtyping constraints have to be explicitly contained in $\bar{\gamma}$ (as are all the other constraints).

Compilers can infer the most general type constraints for a given source; the idea, detailed in [1], is to use fresh type variables everywhere a type is not explicitly given by the programmer, and to generate type constraints only corresponding to required "actions" (that is, if the source contains a field access expression, then a field access constraint is generated and so on). At the JVM level, however, we are not interested to know whether they are the most general or not, as long as they are strong enough. Indeed, developers could use type constraints to enforce particular requirements, not apparent from the source code, if they desire to.

JIT-Compilation

The metarules defining the JIT-compilation judgment are shown in Figure 6; the main judgment $\mathcal{B}^{\mathtt{L}}, \mathcal{M}, \sigma \vdash \mathtt{c} \Rightarrow \mathcal{M}'$ is defined in terms of the judgment

$$\mathcal{B}^{\mathtt{L}}, \mathcal{M}, \sigma \vdash \mathtt{cd^b} \Rightarrow \nu$$

to be read "given the environment of loaded classes $\mathcal{B}^{\mathtt{L}}$, the environment of verified classes $\mathcal{M}$ and the substitution σ, the JIT-compilation of class declaration $\mathtt{cd^b}$ produces the virtual method table ν".

This last judgment is defined in the second and third rules; the former deals with the special case of class `Object`, while the latter handles all the other classes.

For non-`Object` classes, the resulting virtual method table corresponds to table of the superclass $\mathtt{c}'$ updated with methods declared in $\mathtt{c}$: new methods, that is methods that do not override any inherited method, get new slots in the table, while overriding methods reuse the indexes used by the superclasses. This calculation is performed by the auxiliary function *idxMth* discussed below.

$$\frac{\mathcal{B}^{\mathrm{L}}, \mathcal{M}, \sigma \vdash \mathsf{cd}^{\mathsf{b}} \Rightarrow \nu}{\mathcal{B}^{\mathrm{L}}, \mathcal{M}, \sigma \vdash \mathsf{c} \Rightarrow \mathcal{M}[\nu/\mathsf{c}]} \qquad \mathcal{B}^{\mathrm{L}}(\mathsf{c}) = (\_, \mathsf{cd}^{\mathsf{b}})$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathcal{M}, \sigma \vdash \texttt{class Object } \{\} \Rightarrow \emptyset}$$

$$\frac{i \in 1..n \;\; \mathcal{B}^{\mathrm{L}}, \mathsf{c} \vdash \sigma(\mathsf{md}_i^{\mathsf{b}}) \Rightarrow \mathsf{e}_i^{\mathsf{r}}}{\begin{array}{l}\mathcal{B}^{\mathrm{L}}, \mathcal{M}, \sigma \vdash \; \texttt{class } \mathsf{c} \texttt{ extends } \mathsf{c}' \; \{ \; \overline{\mathsf{fd}^{\mathsf{b}}} \;\; \mathsf{md}_1^{\mathsf{b}} \ldots \mathsf{md}_n^{\mathsf{b}} \; \} \Rightarrow \\ \qquad \mathcal{M}(\mathsf{c}')[\mathsf{e}_1^{\mathsf{r}}/idxMth_{\mathcal{B}^{\mathrm{L}}}(\mathsf{c}, \mathsf{md}_1^{\mathsf{b}}), \ldots, \mathsf{e}_n^{\mathsf{r}}/idxMth_{\mathcal{B}^{\mathrm{L}}}(\mathsf{c}, \mathsf{md}_n^{\mathsf{b}})]\end{array}}$$

$$\frac{\mathcal{B}^{\mathrm{L}}, \{\texttt{this} \mapsto 0, \mathsf{x}_1 \mapsto 1, \ldots, \mathsf{x}_n \mapsto n\} \vdash \mathsf{e}^{\mathsf{b}} \Rightarrow \mathsf{e}^{\mathsf{r}}}{\mathcal{B}^{\mathrm{L}}, \mathsf{c} \vdash \mathsf{c}_0 \; \mathsf{m}(\mathsf{c}_1 \; \mathsf{x}_1, \; \ldots, \; \mathsf{c}_n \; \mathsf{x}_n) \; \{ \; \texttt{return } \mathsf{e}^{\mathsf{b}}\texttt{;} \; \} \Rightarrow \mathsf{e}^{\mathsf{r}}}$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \mathsf{x} \Rightarrow \texttt{aload}(n)} \qquad \mathcal{P}(\mathsf{x}) = n$$

$$\frac{\mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \mathsf{e}^{\mathsf{b}} \Rightarrow \mathsf{e}^{\mathsf{r}}}{\mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \mathsf{e}^{\mathsf{b}}[\mathsf{c}.\mathsf{f} \; \mathsf{c}'] \Rightarrow \mathsf{e}^{\mathsf{r}}.\texttt{getfield}(n)} \qquad n = idxFld_{\mathcal{B}^{\mathrm{L}}}(\mathsf{c}, \mathsf{f}, \mathsf{c}')$$

$$\frac{i \in 0..k \;\; \mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \mathsf{e}_i^{\mathsf{b}} \Rightarrow \mathsf{e}_i^{\mathsf{r}}}{\begin{array}{l}\mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \; \mathsf{e}_0^{\mathsf{b}}[\mathsf{c}.\mathsf{m}(\bar{\mathsf{c}})\mathsf{c}'](\mathsf{e}_1^{\mathsf{b}}, \; \ldots, \; \mathsf{e}_k^{\mathsf{b}}) \Rightarrow \\ \qquad \mathsf{e}_0^{\mathsf{r}}.\texttt{invokevirtual}(n, \mathsf{e}_1^{\mathsf{r}}, \ldots, \mathsf{e}_k^{\mathsf{r}})\end{array}} \qquad n = idxMth_{\mathcal{B}^{\mathrm{L}}}(\mathsf{c}, \mathsf{m}, \bar{\mathsf{c}}, \mathsf{c}')$$

$$\frac{i \in 1..n \;\; \mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \mathsf{e}_i^{\mathsf{b}} \Rightarrow \mathsf{e}_i^{\mathsf{r}}}{\mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \texttt{new } [\mathsf{c} \; \_](\mathsf{e}_1^{\mathsf{b}}, \ldots, \; \mathsf{e}_n^{\mathsf{b}}) \Rightarrow \mathsf{c}.\texttt{new}(\mathsf{e}_1^{\mathsf{r}}, \ldots, \mathsf{e}_n^{\mathsf{r}})}$$

$$\frac{\mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \mathsf{e}^{\mathsf{b}} \Rightarrow \mathsf{e}^{\mathsf{r}}}{\mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash (\mathsf{c})\mathsf{e}^{\mathsf{b}} \Rightarrow \mathsf{c}.\texttt{checkcast}(\mathsf{e}^{\mathsf{r}})}$$

$$\frac{\mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \mathsf{e}^{\mathsf{b}} \Rightarrow \mathsf{e}^{\mathsf{r}}}{\mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \ll \mathsf{c}, \mathsf{c}' \gg \mathsf{e}^{\mathsf{b}} \Rightarrow \mathsf{e}^{\mathsf{r}}} \qquad subtype_{\mathcal{B}}(\mathsf{c}', \mathsf{c})$$

$$\frac{\mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \mathsf{e}^{\mathsf{b}} \Rightarrow \mathsf{e}^{\mathsf{r}}}{\mathcal{B}^{\mathrm{L}}, \mathcal{P} \vdash \ll \mathsf{c}, \mathsf{c}' \gg \mathsf{e}^{\mathsf{b}} \Rightarrow \mathsf{c}.\texttt{checkcast}(\mathsf{e}^{\mathsf{r}})} \qquad \neg subtype_{\mathcal{B}}(\mathsf{c}', \mathsf{c})$$

Figure 6: Rewrite rules for compilation.

$$idxFld_{\mathcal{B}}(\mathsf{c},\mathsf{f},\mathsf{c}') = i \text{ if } \begin{cases} allFields_{\mathcal{B}}(\mathsf{c}) = \mathsf{fd}_1^{\mathsf{b}} \ldots \mathsf{fd}_n^{\mathsf{b}} \\ \mathsf{fd}_i^{\mathsf{b}} = \mathsf{c}'\ \mathsf{f} \\ \neg\exists j > i \,:\, \mathsf{fd}_j^{\mathsf{b}} = \mathsf{c}'\ \mathsf{f} \end{cases}$$

$$allFields_{\mathcal{B}}(\texttt{Object}) = \Lambda$$
$$allFields_{\mathcal{B}}(\mathsf{c}) = allFields_{\mathcal{B}}(super_{\mathcal{B}}(\mathsf{c}))\, fields_{\mathcal{B}}(\mathsf{c})$$

$$subtype_{\mathcal{B}}(\mathsf{c},\mathsf{c}') = (\mathsf{c}=\mathsf{c}') \text{ or } \exists n \geq 0 :\ super_{\mathcal{B}}(\mathsf{c}) = \mathsf{c}_1 \ \wedge\ \ldots\ \wedge\ super_{\mathcal{B}}(\mathsf{c}_n) = \mathsf{c}'$$

$$idxMth_{\mathcal{B}}(\mathsf{c},\mathsf{c}_0,\mathsf{m},\mathsf{c}_1 \ldots \mathsf{c}_n) =$$
$$\begin{cases} idxMth_{\mathcal{B}}(super_{\mathcal{B}}(\mathsf{c}),\mathsf{c}_0,\mathsf{m},\mathsf{c}_1 \ldots \mathsf{c}_n) & \text{if defined} \\ \mathtt{max}(\{-1\} \cup \{idxMth_{\mathcal{B}}(super_{\mathcal{B}}(\mathsf{c}),\_,\_,\_)\}) + & \text{otherwise} \\ \qquad \sum_{j=1}^{i} notPresent_{\mathcal{B}}(super_{\mathcal{B}}(\mathsf{c}),\mathsf{md}_j^{\mathsf{b}}) & \\ \quad \text{where:} & \\ \qquad methods_{\mathcal{B}}(\mathsf{c}) = \mathsf{md}_1^{\mathsf{b}} \ldots \mathsf{md}_k^{\mathsf{b}} & \\ \qquad \mathsf{md}_i^{\mathsf{b}} = \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1,\ \ldots,\ \mathsf{c}_n\ \mathsf{x}_n)\ \{\ \mathtt{return}\ \mathsf{e}^{\mathsf{b}};\ \} & \end{cases}$$

$$notPresent_{\mathcal{B}}(\mathsf{c},\mathsf{md}^{\mathsf{b}}) = \begin{cases} 1 & \text{if } idxMth_{\mathcal{B}}(\mathsf{c},\mathsf{md}^{\mathsf{b}}) \text{ undefined} \\ 0 & \text{otherwise} \end{cases}$$

$$idxMth_{\mathcal{B}}(\mathsf{c},\mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1,\ \ldots,\ \mathsf{c}_n\ \mathsf{x}_n)\ \{\ \mathtt{return}\ \mathsf{e}^{\mathsf{b}};\ \}) = idxMth_{\mathcal{B}}(\mathsf{c},\mathsf{c}_0,\mathsf{m},\mathsf{c}_1 \ldots \mathsf{c}_n)$$

Figure 7: Auxiliary function for compilation.

This metarule uses the auxiliary judgment

$$\mathcal{B}^{\mathsf{L}}, \mathsf{c} \vdash \mathsf{md}^{\mathsf{b}} \Rightarrow \mathsf{e}^{\mathsf{r}}$$

to be read "given the environment of loaded classes $\mathcal{B}^{\mathsf{L}}$, inside the class c, the binary method declaration md^{b} is JIT-compiled to runtime expression e^{r}".

This judgment is defined using the judgment

$$\mathcal{B}^{\mathsf{L}}, \mathcal{P} \vdash \mathsf{e}^{\mathsf{b}} \Rightarrow \mathsf{e}^{\mathsf{r}}$$

to be read "given the environment of loaded classes $\mathcal{B}^{\mathsf{L}}$ and the parameter environment $\mathcal{P}$, the binary expression e^{b} is JIT-compiled to the runtime expression e^{r}". The parameter environment $\mathcal{P}$ maps `this` and parameter names into their corresponding position; `this` is treated as the zeroth (implicit) parameter, so its index is 0. In the translation:

- Parameter names and `this` are translated into `aload` instructions.
- Field accesses into `getfield` instructions.
- Method invocations into `invokevirtual` instructions.
- Instance creation expressions into `new` instructions; note that we ignore the type of formal parameters of the invoked constructor because in our model there is only one, synthetic, constructor for each class.
- Cast expressions into `checkcast` instructions.
- Polymorphic cast expressions into either the translation of their subexpression, when we already know that such casts would always succeed, or into `checkcast` instructions otherwise.

The metarules for field accesses and method invocations are straightforward because they depend heavily on a couple of auxiliary functions defined in Figure 7.

The auxiliary function $idxFld_{\mathcal{B}}(\mathsf{c}, \mathsf{f}, \mathsf{c}')$ returns the index of the field named f of type c' in an object of type c in the given binary environment $\mathcal{B}$. If more than one field named f of type c' is declared in c and its superclasses, then the greatest index is returned: this corresponds to choosing the field declared nearest to c.

The auxiliary function $idxMth_{\mathcal{B}}(\mathsf{c}, \mathsf{c}_0, \mathsf{m}, \mathsf{c}_1 \ldots \mathsf{c}_n)$ returns the index of the method named m, whose return type is c_0 and formal parameters have types $\mathsf{c}_1, \ldots, \mathsf{c}_n$ in the virtual method table of class c, in the given binary environment $\mathcal{B}$. This function must take into account that the index of an overriding method must be the same of the overridden method, and that all new methods (that is, methods that do not override any other method) should get indexes that are not used by any superclass.

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\gamma, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathtt{load}(\mathtt{c}, \mathtt{verify}(\gamma, \mathcal{V}))} \quad \begin{array}{l} \mathtt{c} \notin \mathit{def}(\mathcal{B}^{\mathrm{L}}) \\ \gamma \in \{\ \mathtt{c} \leq \_,\ \phi(\mathtt{c}, \_, \_), \\ \qquad \mu(\mathtt{c}, \_, \_, (\_, \_)),\ \kappa(\mathtt{c}, \_, \_)\} \end{array}$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\mathtt{c}_1 \leq \mathtt{c}_2, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathcal{V}} \quad \mathtt{c}_1 \in \mathit{def}(\mathcal{B}^{\mathrm{L}}) \wedge (\mathtt{c}_1 = \mathtt{c}_2 \vee \mathtt{c}_2 = \mathtt{Object})$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\phi(\mathtt{c}, \mathtt{f}, \mathtt{t}), \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathtt{match}(\mathtt{t}, \mathtt{c}', \mathcal{V})} \quad \begin{array}{l} \mathtt{c} \in \mathit{def}(\mathcal{B}^{\mathrm{L}}) \\ \mathtt{c}'\ \mathtt{f} \in \mathit{fields}_{\mathcal{B}^{\mathrm{L}}}(\mathtt{c}) \end{array}$$

$$\frac{}{\begin{array}{l}\mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\mu(\mathtt{c}, \mathtt{m}, \bar{\mathtt{c}}, (\mathtt{t}, \bar{\mathtt{t}})), \mathcal{V}) \rightarrow_{\mathcal{B}} \\ \mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\mathtt{c}_1 \leq \mathtt{c}'_1\ \ldots\ \mathtt{c}_n \leq \mathtt{c}'_n, \mathtt{match}(\mathtt{t}\,\bar{\mathtt{t}}, \mathtt{c}'\,\bar{\mathtt{c}}, \mathcal{V}))\end{array}} \quad \begin{array}{l} \mathtt{c} \in \mathit{def}(\mathcal{B}^{\mathrm{L}}) \\ \mathtt{c}'\ \mathtt{m}(\bar{\mathtt{c}}') \in \mathit{methods}_{\mathcal{B}^{\mathrm{L}}}(\mathtt{c}) \\ \bar{\mathtt{c}} = \mathtt{c}_1 \ldots \mathtt{c}_n \\ \bar{\mathtt{c}}' = \mathtt{c}'_1 \ldots \mathtt{c}'_n \end{array}$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\kappa(\mathtt{Object}, \Lambda, \Lambda), \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathcal{V}} \quad \mathtt{Object} \in \mathit{def}(\mathcal{B}^{\mathrm{L}})$$

$$\frac{}{\begin{array}{l}\mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\kappa(\mathtt{c}, \bar{\mathtt{c}}_{\mathtt{a}}\,\bar{\mathtt{c}}_{\mathtt{b}}, \bar{\mathtt{t}}_{\mathtt{a}}\,\bar{\mathtt{t}}_{\mathtt{b}}), \mathcal{V}) \rightarrow_{\mathcal{B}} \\ \mathcal{B}^{\mathrm{L}}, \mathtt{match}(\ \bar{\mathtt{t}}_{\mathtt{b}}, \mathtt{c}'_1 \ldots \mathtt{c}'_m, \\ \qquad \mathtt{verify}(\mathtt{c}_{n+1} \leq \mathtt{c}'_1\ \ldots \\ \qquad \mathtt{c}_{n+m} \leq \mathtt{c}'_m\ \kappa(\mathtt{c}', \bar{\mathtt{c}}_{\mathtt{a}}, \bar{\mathtt{t}}_{\mathtt{a}}), \mathcal{V}))\end{array}} \quad \begin{array}{l} \mathtt{c} \in \mathit{def}(\mathcal{B}^{\mathrm{L}}) \\ \bar{\mathtt{c}}_{\mathtt{a}} = \mathtt{c}_1 \ldots \mathtt{c}_n \\ \bar{\mathtt{c}}_{\mathtt{b}} = \mathtt{c}_{n+1} \ldots \mathtt{c}_{n+m} \\ \bar{\mathtt{t}}_{\mathtt{a}} = \mathtt{t}_1 \ldots \mathtt{t}_n \\ \bar{\mathtt{t}}_{\mathtt{b}} = \mathtt{t}_{n+1} \ldots \mathtt{t}_{n+m} \\ \mathtt{c}'_1\ \mathtt{f}_1 \ldots \mathtt{c}'_m\ \mathtt{f}_m = \mathit{fields}_{\mathcal{B}^{\mathrm{L}}}(\mathtt{c}) \\ \mathtt{c}' = \mathit{super}_{\mathcal{B}^{\mathrm{L}}}(\mathtt{c}) \end{array}$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\mathtt{c} \sim \mathtt{c}', \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathcal{V}} \quad \mathtt{c}' \in \{\mathtt{c}, \mathtt{Object}\}$$

$$\frac{}{\begin{array}{l}\mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\mathtt{c} \sim \mathtt{c}', \mathcal{V}) \rightarrow_{\mathcal{B}} \\ \mathcal{B}^{\mathrm{L}}, \mathtt{verifyEither}(\mathtt{verify}(\mathtt{c} \leq \mathtt{c}', \emptyset), \mathtt{verify}(\mathtt{c}' \leq \mathtt{c}, \emptyset), \mathcal{V})\end{array}}$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{match}(\alpha, \mathtt{c}, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathcal{V}[\alpha \mapsto \mathtt{c}]} \qquad \frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{match}(\mathtt{c}, \mathtt{c}, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathcal{V}}$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{match}(\mathtt{c}, \mathtt{c}', \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathtt{VerifyError}} \quad \mathtt{c} \neq \mathtt{c}'$$

$$\mathit{fields}_{\mathcal{B}}(\mathtt{c}) = \mathit{fields}(\mathit{classDeclaration}(\mathcal{B}(\mathtt{c})))$$
$$\mathit{fields}(\mathtt{class\ c\ extends\ c'\ \{}\ \overline{\mathtt{fd}^{\mathtt{b}}}\ \overline{\mathtt{md}^{\mathtt{b}}}\ \mathtt{\}}) = \overline{\mathtt{fd}^{\mathtt{b}}}$$
$$\mathit{methods}_{\mathcal{B}}(\mathtt{c}) = \mathit{methods}(\mathit{classDeclaration}(\mathcal{B}(\mathtt{c})))$$
$$\mathit{methods}(\mathtt{class\ c\ extends\ c'\ \{}\ \overline{\mathtt{fd}^{\mathtt{b}}}\ \overline{\mathtt{md}^{\mathtt{b}}}\ \mathtt{\}}) = \overline{\mathtt{md}^{\mathtt{b}}}$$

Figure 8: Rewrite rules for verifying constraints.

Constraint Verification

Rewriting rules for constraints verification are given in Figure 8. The first rule ensures that a class is loaded before we try to check any property about it. For instance, if we need to check whether class `c` provides a certain field and `c` has not been loaded yet, then we load it and postpone the check until `c` has been loaded.

The second rule checks the subtype constraint $\mathtt{c}_1 \leq \mathtt{c}_2$ when $\mathtt{c}_1$ has already been loaded and $\mathtt{c}_2$ is equal to $\mathtt{c}_1$ or to `Object`. If these conditions are met, then the constraint holds (without any substitution) so we can discard it and go on. Of course, this is a very special case: what if the side conditions are not met? If they are not met because $\mathtt{c}_1$ has not been loaded, then the first rule would come into play; however, we need something else for all other cases. If $\mathtt{c}_1 \neq \mathtt{c}_2$ and $\mathtt{c}_2 \neq \mathtt{Object}$ then we have two possibilities: if $\mathtt{c}_1$ is equal to `Object` then the verification should fail (since `Object` is the only subtype of itself) otherwise we should check whether the constraint holds taking the superclass of $\mathtt{c}_1$ in place of $\mathtt{c}_1$. Both situations are covered by the error and propagation rules shown in Figure 9 and described below.

The third rule checks whether class `c` provides a field named `f`. As it happens for the subtype constraint checking just described, this rule deals only with the case when `c` directly declares the field `f`, while propagation/error rules in Figure 9 handle the other cases. If `c` declares a field named `f` of type $\mathtt{c}'$ the verification of the constraint is rewritten into a `match` action, matching the expected type `t` (usually a type variable) with the declared type $\mathtt{c}'$. As shown in the lower part of the figure, matching a type variable α with a class name `c` always succeeds and corresponds to replace all occurrences of α with `c` in $\mathcal{V}$. Matching a class name `c` with another class name $\mathtt{c}'$ always fails (except for the trivial case $\mathtt{c} = \mathtt{c}'$) because it corresponds to the fact that a particular class was expected and a different one has been found in the system.

The fourth rule checks whether a class `c` provides a method named `m` with the right number of parameters. As before, rules in Figure 9 handle the other cases. If the method is found the constraint is satisfied as long as the actual argument types are subtypes of the formal ones, and the expected formal parameter and return type match the actual ones[12].

The fifth and sixth rules check constructor constraints. The former encodes that class `Object` provides just the default constructor, while the latter deals with all other classes. Again, error cases are dealt in Figure 9. In our model all classes implicitly provide a constructor that receives the initial values for all fields (that is, the declared and the inherited fields) of the class. For this reason, the parameter types of a constructor always consists of the sequence of the types of inherited fields followed by the types of the declared fields. So, if class `c` declares m fields, whose

[12]There might seem to be a duplication here: why do we need to match the formal parameter types with, presumably, type variables when we already know that the actual parameter types are subtypes of the formal ones? The point is that we inherited the type constraints from [1] where they need this distinction to handle a peculiarity of the standard Java binary format.

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\Lambda, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathcal{V}}$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\gamma\bar{\gamma}, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\gamma, \mathtt{verify}(\bar{\gamma}, \mathcal{V}))} \quad \bar{\gamma} \neq \Lambda$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{match}(\Lambda, \Lambda, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathcal{V}}$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{match}(\mathsf{c}\bar{\mathsf{c}}, \mathsf{t}\bar{\mathsf{t}}, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathtt{match}(\mathsf{c}, \mathsf{t}, \mathtt{match}(\bar{\mathsf{c}}, \bar{\mathsf{t}}, \mathcal{V}))} \quad \bar{\mathsf{c}} \neq \Lambda$$

$$\frac{\mathcal{B}^{\mathrm{L}} \vdash [\mathsf{c}]^{Super}}{\mathcal{B}^{\mathrm{L}}, \mathtt{verify}([\mathsf{c}]^{Super}, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathtt{verify}([super_{\mathcal{B}^{\mathrm{L}}}(\mathsf{c})]^{Super}, \mathcal{V})} \quad \begin{array}{l} \mathsf{c} \in def(\mathcal{B}^{\mathrm{L}}) \\ \mathsf{c} \neq \mathtt{Object} \end{array}$$

$$[\cdot]^{Super} ::= [\cdot] \leq \mathsf{c} \;\mid\; \phi([\cdot], \_, \_) \;\mid\; \mu([\cdot], \_, \_, \_)$$

$$\frac{}{\mathcal{B}^{\mathrm{L}} \vdash \mathsf{c}_1 \leq \mathsf{c}_2} \quad \begin{array}{l} \mathsf{c}_1 \neq \mathsf{c}_2 \\ \mathsf{c}_2 \neq \mathtt{Object} \end{array}$$

$$\frac{}{\mathcal{B}^{\mathrm{L}} \vdash \phi(\mathsf{c}, \mathsf{f}, \_)} \quad \_\ \mathsf{f} \notin fields_{\mathcal{B}^{\mathrm{L}}}(\mathsf{c})$$

$$\frac{}{\mathcal{B}^{\mathrm{L}} \vdash \mu(\mathsf{c}, \mathsf{m}, \mathsf{c}_1 \ldots \mathsf{c}_n, (\_, \_))} \quad \_\ \mathsf{m}(\mathsf{c}'_1 \ldots \mathsf{c}'_n) \notin methods_{\mathcal{B}^{\mathrm{L}}}(\mathsf{c})$$

$$\frac{}{\mathcal{B}^{\mathrm{L}}, \mathtt{verify}(\gamma, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathrm{L}}, \mathtt{VerifyError}} \quad \begin{array}{l} \mathtt{Object} \in def(\mathcal{B}^{\mathrm{L}}) \\ \mathsf{c} \neq \mathtt{Object} \\ n, m > 0 \\ \gamma \in \{\ \mathtt{Object} \leq \mathsf{c},\ \phi(\mathtt{Object}, \_, \_), \\ \qquad \mu(\mathtt{Object}, \_, \_, (\_, \_)), \\ \qquad \kappa(\mathtt{Object}, \mathsf{c}_1 \ldots \mathsf{c}_n, \mathsf{t}_1 \ldots \mathsf{t}_m)\} \end{array}$$

Figure 9: Propagation and error rules for verifying constraints.

$$\frac{\mathcal{B}^{\mathsf{L}}_1, \mathcal{V}_1 \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \mathcal{V}_4}{\mathcal{B}^{\mathsf{L}}_1, \texttt{verifyEither}(\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \texttt{verifyEither}(\mathcal{V}_4, \mathcal{V}_2, \mathcal{V}_3)}$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \texttt{verifyEither}(\sigma, \mathcal{V}_2, \mathcal{V}_3) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \sigma(\mathcal{V}_3)}$$

$$\frac{\mathcal{B}^{\mathsf{L}}_1, \mathcal{V}_2 \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \mathcal{V}_4}{\mathcal{B}^{\mathsf{L}}_1, \texttt{verifyEither}(\epsilon, \mathcal{V}_2, \mathcal{V}_3) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \texttt{verifyEither}(\epsilon, \mathcal{V}_4, \mathcal{V}_3)}$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \texttt{verifyEither}(\epsilon, \epsilon', \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \epsilon'}$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \texttt{verifyEither}(\epsilon, \sigma, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \sigma(\mathcal{V})}$$

Figure 10: Contextual closure for execution of `verifyEither`.

$$\frac{\begin{array}{l}\Lambda, \texttt{verify}(\bar{\gamma}, \emptyset) \rightarrow^{+}_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \sigma \\ \mathcal{B}^{\mathsf{L}}, \emptyset \vdash \sigma(\mathsf{e}^{\mathsf{b}}) \Rightarrow \mathsf{e}^{\mathsf{r}}\end{array}}{\Lambda, \Lambda, \texttt{bootstrap}(\bar{\gamma}, \mathsf{e}^{\mathsf{b}}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \Lambda, \mathsf{e}^{\mathsf{r}}} \quad \begin{array}{l}\mathit{wellFormed}(\bar{\gamma}) \\ \bar{\gamma}, \emptyset \vdash \mathsf{e}^{\mathsf{b}} : \_\end{array}$$

$$\frac{\Lambda, \texttt{verify}(\bar{\gamma}, \emptyset) \rightarrow^{+}_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \epsilon}{\Lambda, \Lambda, \texttt{bootstrap}(\bar{\gamma}, \mathsf{e}^{\mathsf{b}}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \Lambda, \epsilon} \quad \begin{array}{l}\mathit{wellFormed}(\bar{\gamma}) \\ \bar{\gamma}, \emptyset \vdash \mathsf{e}^{\mathsf{b}} : \_\end{array}$$

$$\frac{}{\Lambda, \Lambda, \texttt{bootstrap}(\bar{\gamma}, \mathsf{e}^{\mathsf{b}}) \rightsquigarrow_{\mathcal{B}} \Lambda, \Lambda, \texttt{VerifyError}} \quad \neg \mathit{wellFormed}(\bar{\gamma}) \vee \bar{\gamma}, \emptyset \nvdash \mathsf{e}^{\mathsf{b}} : \_$$

Figure 11: Rewrite rules for bootstrapping the system.

types are $\mathsf{c}'_1 \ldots \mathsf{c}'_m$, we match these types with the expected ones[13], verify that each actual argument is assignable to the corresponding field and, finally, verify that the remaining arguments can be applied to the superclass.

The seventh and eighth rules check whether the class names c and c' are comparable. They are trivially comparable when they are equal or one of them is `Object` (seventh rule), otherwise we need to check whether one of them is a subtype of the other. This is obtained through the action `verifyEither`, whose execution rewriting rules are given in Figure 10.

Bootstrapping

Now that we have seen all the ingredients, we can describe the rules for `bootstrap` given in Figure 11. Execution starts with the special expression `bootstrap` from the empty environments of loaded and verified classes:

$$\Lambda, \Lambda, \texttt{bootstrap}(\bar{\gamma}, \mathsf{e}^{\mathsf{b}}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{M}, \ldots$$

The binary expression e^{b} corresponds to the code of the `main` method and $\bar{\gamma}$ contains its type constraints. In the first rule of the figure, the constraints $\bar{\gamma}$ are well-formed (first side-condition), the expression e^{b} is compliant with them (second side-condition) and verification succeeds producing the substitution σ (premise). In these setting we can use the JIT-compilation judgment to obtain the runtime expression to run.

The other two rules deal with error case: if verification fails with an exception ϵ, second rule, then ϵ is propagated so it becomes the result of the computation. If the constraint $\bar{\gamma}$ are not well-formed or the expression e^{b} is not compliant with $\bar{\gamma}$, third rule, then the execution immediately ends with the exception `VerifyError`.

Finally, the rules describing normal execution, abnormal execution (that is, exception throwing) and standard closures are given in Figures 12 and 13.

These rules are quite standard, with the notable exception of the third rule of Figure 12: this rules triggers the verification of class `c`, if it has not verified yet, before allowing the execution of the `new` expression to create instances of `c`.

Results

To show that the execution does not get stuck we first define a typing judgment on runtime expressions and a notion of consistency between loaded classes and virtual tables. Then, we use the standard approach of proving subject reduction, Theorem 5, and progress, Theorem 6.

The judgment $\mathcal{B}^{\mathsf{L}}, \Pi \vdash \mathsf{e}^{\mathsf{r}} : \mathsf{c}$, defined in Figure 14, means "given the binary environment of loaded classes $\mathcal{B}$, and parameter environment Π, the runtime expression e^{r} has type c"; the parameter environment, which maps parameter positions to their types, is necessary only to type method bodies.

A binary environment $\mathcal{B}$ and a set of virtual tables $\mathcal{M}$ are consistent when the tables map indexes into method bodies having a type which is a subtype of the declared return type; this is formalized by the judgment $\mathcal{B} \vdash \mathcal{M}$, also shown in Figure 14.

Before stating the main theorems, we introduce some auxiliary lemmas and theorems. The proofs are only sketched.

[13] Same reason of the method constraints, see the previous footnote for further details.

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathcal{M}, \texttt{new c}(v_1, \ldots, v_n).\texttt{getfield}(i) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{M}, v_i}$$

$$\frac{}{\begin{array}{l}\mathcal{B}^{\mathsf{L}}, \mathcal{M}, \texttt{new c}(\bar{v}).\texttt{invokevirtual}(i, v_1, \ldots, v_n) \rightsquigarrow_{\mathcal{B}} \\ \mathcal{B}^{\mathsf{L}}, \mathcal{M}, \mathcal{M}(\mathsf{c})(i)[\texttt{new c}(\bar{v})/\texttt{aload}(0), v_1/\texttt{aload}(1), \ldots, v_n/\texttt{aload}(n)]\end{array}}$$

$$\frac{}{\begin{array}{l}\mathcal{B}^{\mathsf{L}}, \mathcal{M}, \mathsf{c}.\texttt{new}(v_1, \ldots, v_n) \rightsquigarrow_{\mathcal{B}} \\ \mathcal{B}^{\mathsf{L}}, \mathcal{M}, \texttt{verifyCls}(\mathsf{c}, \mathsf{c}.\texttt{new}(v_1, \ldots, v_n))\end{array}} \quad \mathsf{c} \notin \mathit{def}(\mathcal{M})$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathcal{M}, \mathsf{c}.\texttt{new}(v_1, \ldots, v_n) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{M}, \texttt{new c}(v_1 \ldots v_n)} \quad \mathsf{c} \in \mathit{def}(\mathcal{M})$$

$$\frac{}{\begin{array}{l}\mathcal{B}^{\mathsf{L}}, \mathcal{M}, \texttt{new c}(\bar{v}).\texttt{checkcast}(\mathsf{c}') \rightsquigarrow_{\mathcal{B}} \\ \mathcal{B}^{\mathsf{L}}, \mathcal{M}, \texttt{new c}(\bar{v})\end{array}} \quad \mathit{subtype}_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}, \mathsf{c}')$$

$$\frac{}{\begin{array}{l}\mathcal{B}^{\mathsf{L}}, \mathcal{M}, \texttt{new c}(\bar{v}).\texttt{checkcast}(\mathsf{c}') \rightsquigarrow_{\mathcal{B}} \\ \mathcal{B}^{\mathsf{L}}, \mathcal{M}, \texttt{ClassCastException}\end{array}} \quad \neg \mathit{subtype}_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}, \mathsf{c}')$$

Figure 12: Rewrite rules for execution.

$$[\cdot]^{Exp} ::= [\cdot].\texttt{getfield}(n) \mid \mathsf{c}.\texttt{new}(\bar{v}, [\cdot], \overline{\mathsf{e}^{\mathsf{r}}}) \mid [\cdot].\texttt{checkcast}(\mathsf{c}) \mid [\cdot].\texttt{invokevirtual}(n, \overline{\mathsf{e}^{\mathsf{r}}}) \mid v.\texttt{invokevirtual}(n, \bar{v}, [\cdot], \overline{\mathsf{e}^{\mathsf{r}}})$$

$$\frac{\mathcal{B}_1^{\mathsf{L}}, \mathcal{M}_1, \mathsf{e}_1^{\mathsf{r}} \rightsquigarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{M}_2, \mathsf{e}_2^{\mathsf{r}}}{\mathcal{B}_1^{\mathsf{L}}, \mathcal{M}_1, [\mathsf{e}_1^{\mathsf{r}}]^{Exp} \rightsquigarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{M}_2, [\mathsf{e}_2^{\mathsf{r}}]^{Exp}} \qquad \frac{\mathcal{B}_1^{\mathsf{L}}, \mathcal{M}_1, \mathsf{e}^{\mathsf{r}} \rightsquigarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{M}_2, \epsilon}{\mathcal{B}_1^{\mathsf{L}}, \mathcal{M}_1, [\mathsf{e}^{\mathsf{r}}]^{Exp} \rightsquigarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{M}_2, \epsilon}$$

Figure 13: Contextual closure for regular and abnormal execution.

$$\frac{i \in 1..k \quad \mathcal{B}^{\mathsf{L}}, \Pi \vdash v_i : \mathsf{c}_i}{\mathcal{B}^{\mathsf{L}}, \Pi \vdash \texttt{new}\ \mathsf{c}(v_1, \ldots, v_k) : \mathsf{c}} \qquad \begin{array}{l} allFields_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}) = \mathsf{c}'_1\ \mathsf{f}_1\,; \ldots \mathsf{c}'_k\ \mathsf{f}_k\,; \\ i \in 1..k\ subtype_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}_i, \mathsf{c}'_i) \end{array}$$

$$\frac{\mathcal{B}^{\mathsf{L}}, \Pi \vdash \mathsf{e}^{\mathsf{r}} : \mathsf{c}}{\mathcal{B}^{\mathsf{L}}, \Pi \vdash \mathsf{e}^{\mathsf{r}}.\texttt{getfield}(n) : \mathsf{c}'} \qquad \begin{array}{l} allFields_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}) = \mathsf{fd}^{\mathsf{b}}_1 \ldots \mathsf{fd}^{\mathsf{b}}_k \\ \mathsf{fd}^{\mathsf{b}}_n = \mathsf{c}'\ \mathsf{f}\,; \end{array}$$

$$\frac{i \in 0..k \quad \mathcal{B}^{\mathsf{L}}, \Pi \vdash \mathsf{e}^{\mathsf{r}}_i : \mathsf{c}_i}{\mathcal{B}^{\mathsf{L}}, \Pi \vdash \mathsf{e}^{\mathsf{r}}_0.\texttt{invokevirtual}(n, \mathsf{e}^{\mathsf{r}}_1 \ldots \mathsf{e}^{\mathsf{r}}_k) : \mathsf{c}'} \qquad \begin{array}{l} n = idxMth_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}_0, \mathsf{c}', \mathsf{m}, \mathsf{c}'_1 \ldots \mathsf{c}'_k) \\ i \in 1..k\ subtype_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}_i, \mathsf{c}'_i) \end{array}$$

$$\frac{i \in 1..k \quad \mathcal{B}^{\mathsf{L}}, \Pi \vdash \mathsf{e}^{\mathsf{r}}_i : \mathsf{c}_i}{\mathcal{B}^{\mathsf{L}}, \Pi \vdash \mathsf{c}.\texttt{new}(\mathsf{e}^{\mathsf{r}}_1, \ldots, \mathsf{e}^{\mathsf{r}}_k) : \mathsf{c}} \qquad \begin{array}{l} allFields_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}) = \mathsf{c}'_1\ \mathsf{f}_1\,; \ldots \mathsf{c}'_k\ \mathsf{f}_k\,; \\ i \in 1..k\ subtype_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}_i, \mathsf{c}'_i) \end{array}$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \Pi \vdash \texttt{aload}(n) : \mathsf{c}} \qquad \Pi(n) = \mathsf{c}$$

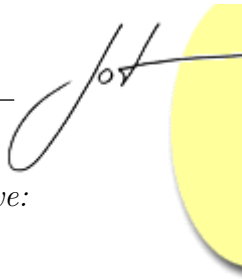
$$\frac{\mathcal{B}^{\mathsf{L}}, \Pi \vdash \mathsf{e}^{\mathsf{r}} : \_}{\mathcal{B}^{\mathsf{L}}, \Pi \vdash \mathsf{e}^{\mathsf{r}}.\texttt{checkcast}(\mathsf{c}) : \mathsf{c}}$$

$$\frac{\mathcal{B}^{\mathsf{L}}, \Pi \vdash \mathsf{e}^{\mathsf{r}} : \mathsf{c}}{\mathcal{B}^{\mathsf{L}}, \Pi \vdash \texttt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}) : \mathsf{c}}$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \Pi \vdash \texttt{bootstrap}(\_, \_) : \texttt{Object}}$$

$$\mathcal{B} \vdash \mathcal{M} \Leftrightarrow \left\{ \begin{array}{l} \forall \mathsf{c}, n : \mathcal{M}(\mathsf{c}) = \nu,\ \nu(n) = \mathsf{e}^{\mathsf{r}} \\ \quad n = idxMth_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}, \mathsf{c}_{\texttt{res}}, \mathsf{m}, \mathsf{c}_1 \ldots \mathsf{c}_k) \\ \quad \forall \mathsf{c}' : subtype_{\mathcal{B}}(\mathsf{c}', \mathsf{c}) : \\ \quad\quad \mathcal{B}, \{0 \mapsto \mathsf{c}', 1 \mapsto \mathsf{c}_1, \ldots, k \mapsto \mathsf{c}_k\} \vdash \mathcal{M}(\mathsf{c}')(n) : \mathsf{c}'_{\texttt{res}} \\ \quad\quad subtype_{\mathcal{B}}(\mathsf{c}'_{\texttt{res}}, \mathsf{c}_{\texttt{res}}) \end{array} \right.$$

Figure 14: Runtime typechecking and consistency.



Lemma 1 *If* $allFields_{\mathcal{B}}(\mathsf{c}) = \mathsf{c}_1\ \mathsf{f}_1\,; \ldots \mathsf{c}_i\ \mathsf{f}_i\,; \ldots$, *then* $\forall \mathsf{c}' : subtype_{\mathcal{B}}(\mathsf{c}', \mathsf{c})$ *we have:* $allFields_{\mathcal{B}}(\mathsf{c}') = \mathsf{c}_1\ \mathsf{f}_1\,; \ldots \mathsf{c}_i\ \mathsf{f}_i\,; \ldots$.

Proof By definition, see Figure 7. □

Lemma 2 $\forall \mathsf{c}, \mathsf{c}' : subtype_{\mathcal{B}}(\mathsf{c}, \mathsf{c}')$, *if* $idxMth_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}, \mathsf{c}_0, \mathsf{m}, \mathsf{c}_1 \ldots \mathsf{c}_n) = i$ *then* $idxMth_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}', \mathsf{c}_0, \mathsf{m}, \mathsf{c}_1 \ldots \mathsf{c}_n) = i$. *Moreover, if* $idxMth_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c}', \mathsf{c}'_0, \mathsf{m}', \mathsf{c}'_1 \ldots \mathsf{c}'_n) = j$ *then*

$$i = j \Leftrightarrow \begin{cases} \mathsf{c}_0 = \mathsf{c}'_0 \ldots \mathsf{c}_n = \mathsf{c}'_n \\ \mathsf{m} = \mathsf{m}' \end{cases}$$

Proof By definition, see Figure 7. □

Theorem 1 *If* $\mathcal{B}, \{0 \mapsto \mathsf{c}_0, \ldots, n \mapsto \mathsf{c}_n\} \vdash \mathsf{e}^{\mathsf{r}} : \mathsf{c}$
$i \in 0..n\ \mathcal{B}, \emptyset \vdash \mathsf{e}^{\mathsf{r}}_i : \mathsf{c}'_i$ *s.t.* $subtype_{\mathcal{B}}(\mathsf{c}'_i, \mathsf{c}_i)$
then $\mathcal{B}, \emptyset \vdash \mathsf{e}^{\mathsf{r}}[\mathsf{e}^{\mathsf{r}}_0/\texttt{aload}(0), \ldots, \mathsf{e}^{\mathsf{r}}_n/\texttt{aload}(n)] : \mathsf{c}''$ *and* $subtype_{\mathcal{B}}(\mathsf{c}'', \mathsf{c}')$.

Proof By induction on the derivation of the typing judgment $\mathcal{B}, \{0 \mapsto \mathsf{c}_0, \ldots, n \mapsto \mathsf{c}_n\} \vdash \mathsf{e}^{\mathsf{r}} : \mathsf{c}$ (metarules in Figure 14), using Lemmas 1 and 2. □

Lemma 3 *If* $\mathcal{B}^{\mathsf{L}}_1 \subseteq \mathcal{B}$, *then either*

- $\mathcal{B}^{\mathsf{L}}_1, \texttt{load}(\mathsf{c}, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \mathcal{V}$ *and* $\mathcal{B}^{\mathsf{L}}_2(\mathsf{c}) = \mathcal{B}(\mathsf{c})$, *or*
- $\mathcal{B}^{\mathsf{L}}_1, \texttt{load}(\mathsf{c}, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \epsilon$

Proof By definition, see Figure 3. □

Lemma 4 *If* $\mathcal{B}^{\mathsf{L}}_1 \subseteq \mathcal{B}$ *and* $\mathcal{B}^{\mathsf{L}}_1, \mathcal{V}_1 \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \mathcal{V}_2$ *then* $\mathcal{B}^{\mathsf{L}}_1 \subseteq \mathcal{B}^{\mathsf{L}}_2 \subseteq \mathcal{B}$.

Proof Trivial. □

Lemma 5 *If* $\mathcal{B}^{\mathsf{L}}_1 \subseteq \mathcal{B}$ *and* $\mathcal{B}^{\mathsf{L}}_1, \mathcal{M}_1, \mathsf{e}^{\mathsf{r}}_1 \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \mathcal{M}_2, \mathsf{e}^{\mathsf{r}}_2$ *then* $\mathcal{B}^{\mathsf{L}}_1 \subseteq \mathcal{B}^{\mathsf{L}}_2 \subseteq \mathcal{B}$ *and* $\mathcal{M}_1 \subseteq \mathcal{M}_2$. *Moreover, if* $\mathcal{M}_1(\mathsf{c}) = \nu$ *then* $\mathcal{M}_2(\mathsf{c}) = \nu$.

Proof Trivial. □

Theorem 2 *If* $wellFormed(\bar{\gamma})$, *then either*

- $\mathcal{B}^{\mathsf{L}}_1, \texttt{verify}(\bar{\gamma}, \emptyset) \rightarrow^{+}_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \sigma$ *or*
- $\mathcal{B}^{\mathsf{L}}_1, \texttt{verify}(\bar{\gamma}, \emptyset) \rightarrow^{+}_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \epsilon$

Proof The well-formedness of $\bar{\gamma}$ implies that input parameters of γ_i are contained in the output parameters of $\gamma_1 \ldots \gamma_{i-1}$ so verification cannot get stuck. Moreover, constraints are removed from the term when they are processed, so the reduction must eventually terminate (into either a substitution or an exception). See Lemma 3 and Figures 8, 9 and 10. □

Theorem 3 *If* $\mathcal{B}_1^{\mathsf{L}}, \mathtt{verify}(\bar{\gamma}, \emptyset) \rightarrow_{\mathcal{B}}^{+} \mathcal{B}_2^{\mathsf{L}}, \sigma$ *then* $\sigma(\bar{\gamma})$ *are ground constraints satisfied in* $\mathcal{B}_2^{\mathsf{L}}$.

Proof By definition of single constraint cases $\mathcal{B}^{\mathsf{L}}, \mathtt{verify}(\gamma, \mathcal{V}) \rightarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \_$, action `match` and propagation rules; see Figures 8, 9 and 10. □

Theorem 4 *If*

$$\begin{array}{l} \mathcal{B}_1^{\mathsf{L}} \vdash \mathcal{M}_1 \\ (\mathsf{cd}^{\mathsf{b}}, \bar{\gamma}) = \mathcal{B}_1^{\mathsf{L}}(\mathsf{c}) \\ \mathtt{readyTBV}(\mathsf{c}, \mathcal{B}_1^{\mathsf{L}}, \mathcal{M}_1) \\ \mathtt{wellFormedAndCompliant}(\bar{\gamma}, \mathsf{cd}^{\mathsf{b}}) \\ \mathcal{B}_1^{\mathsf{L}}, \mathtt{verify}(\bar{\gamma}, \emptyset) \rightarrow_{\mathcal{B}}^{+} \mathcal{B}_2^{\mathsf{L}}, \sigma \end{array}$$

then

$$\begin{array}{l} \mathcal{B}_2^{\mathsf{L}}, \mathcal{M}_1, \sigma \vdash \mathsf{c} \Rightarrow \mathcal{M}_2 \\ \mathcal{B}_2^{\mathsf{L}} \vdash \mathcal{M}_2 \end{array}$$

Proof By compliance and using Theorem 3 it is easy to see that JIT-compilation is defined. The requirements for $\mathcal{B}_2^{\mathsf{L}} \vdash \mathcal{M}_2$ are, by lemmas 4 and 5, met for all classes already in $\mathcal{M}_1$. By Theorem 1 they are met for inherited methods too, so it remains to check only new and overridden methods introduced by class `c`.

For all method declarations of class `c`,

$$\mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1,\ \ldots,\ \mathsf{c}_n\ \mathsf{x}_n)\{\mathtt{return}\ \mathsf{e}^{\mathsf{b}};\} \in \mathsf{cd}^{\mathsf{b}}$$

we know, by hypothesis, that:

- $\bar{\gamma}, \{\mathtt{this} \mapsto \mathsf{c}, \mathsf{x}_1 \mapsto \mathsf{c}_1, \ldots, \mathsf{x}_n \mapsto \mathsf{c}_n\} \vdash \mathsf{e}^{\mathsf{b}} : \mathsf{t}$
- $\bar{\gamma} \vdash \mathsf{t} \leq \mathsf{c}_0$

and, by Theorem 3, $\mathit{subtype}_{\mathcal{B}_2^{\mathsf{L}}}(\sigma(\mathsf{t}), \mathsf{c}_0)$ holds.

Moreover, by induction on typing of binary expressions, considering figures 5, 6 and 14, it can be shown that if $\bar{\gamma}, \{\mathtt{this} \mapsto \mathsf{c}, \mathsf{x}_1 \mapsto \mathsf{c}_1, \ldots, \mathsf{x}_n \mapsto \mathsf{c}_n\} \vdash \mathsf{e}_2^{\mathsf{b}} : \mathsf{t}''$ then $\mathcal{B}_2^{\mathsf{L}}, \{\mathtt{this} \mapsto 0, \mathsf{x}_1 \mapsto 1, \ldots, \mathsf{x}_n \mapsto n\} \vdash \mathsf{e}_2^{\mathsf{b}} \Rightarrow \mathsf{e}_2^{\mathsf{r}}$ and $\mathcal{B}_2^{\mathsf{L}}, \{0 \mapsto \mathsf{c}, 1 \mapsto \mathsf{c}_1, \ldots, n \mapsto \mathsf{c}_n\} \vdash \mathsf{e}_2^{\mathsf{r}} : \mathsf{c}''$ where c'' is such that $\mathit{subtype}_{\mathcal{B}_2^{\mathsf{L}}}(\mathsf{c}'', \sigma(\mathsf{t}''))$ holds. This reasoning, applied to method bodies, proves that all requirements for $\mathcal{B}_2^{\mathsf{L}} \vdash \mathcal{M}_2$ are met. □

Theorem 5 (Subject Reduction) *If*

$$\begin{array}{l}\mathcal{B}_1^{\mathsf{L}}, \Pi \vdash \mathsf{e}_1^{\mathsf{r}} : c_1\\ \mathcal{B}_1^{\mathsf{L}} \vdash \mathcal{M}_1\\ \mathcal{B}_1^{\mathsf{L}}, \mathcal{M}_1, \mathsf{e}_1^{\mathsf{r}} \rightsquigarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{M}_2, \mathsf{e}_2^{\mathsf{r}}\end{array}$$

then, $\mathsf{e}_2^{\mathsf{r}}$ *is an exception or*

$$\begin{array}{l}\mathcal{B}_2^{\mathsf{L}}, \Pi \vdash \mathsf{e}_2^{\mathsf{r}} : c_2\\ \mathit{subtype}_{\mathcal{B}_2^{\mathsf{L}}}(c_2, c_1)\\ \mathcal{B}_2^{\mathsf{L}} \vdash \mathcal{M}_2\end{array}$$

Proof By induction on the reduction rules defining executions (Figures 12, 13 and 11) using Theorem 1 and Theorem 4. □

Theorem 6 (Progress) *If* $\mathcal{B}_1^{\mathsf{L}} \vdash \mathcal{M}_1$, $\mathsf{e}_1^{\mathsf{r}}$ *is neither a value* v *nor an exception* ϵ, *and* $\mathcal{B}_1^{\mathsf{L}}, \emptyset \vdash \mathsf{e}_1^{\mathsf{r}} : \_$ *then*

$$\mathcal{B}_1^{\mathsf{L}}, \mathcal{M}_1, \mathsf{e}_1^{\mathsf{r}} \rightsquigarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{M}_2, \mathsf{e}_2^{\mathsf{r}}$$

Proof By induction on typing rules and using Theorem 2 for action rewriting. □

6 IMPLEMENTATION ISSUES

As briefly mentioned in Section 2, running polymorphic bytecode, as opposed to transforming polymorphic bytecode into standard bytecode at the verification stage, would substantially complicate the implementation of a JVM and it would make very difficult to reuse an existing implementation.

In this section we show some examples of why polymorphic bytecode is more difficult to handle. Let us start discussing method invocation instructions.

The low level code for invoking a method depends on what kind of method is to be invoked. Virtual method invocations typically use an indirection (and a check for nullity), static method invocations do not. A (non-final) private instance method is treated differently from a (non-final) public instance method because the former cannot be overridden. The same reasoning applies, of course, between final and non-final instance methods. All these considerations are invisible at source level, but they matter at binary level. For these reasons, standard Java compilers produce different instructions for handling these cases (namely, `invokevirtual`, `invokestatic` and `invokespecial`).

When sources are compiled in total isolation, compilers cannot, of course, predict which kind of method will be invoked[14]. This suggests that we should probably encode any method invocation using a single generic JVM instruction, at least until we discover, *during execution*, the specific kind of each method invocation (then

[14]Except for few special cases; for instance when a private method is invoked and its parameter types match exactly the argument types.

we could replace the generic instruction with a specific one). Of course, it is more efficient to know beforehand which low level actions are to be taken to execute an invocation. Furthermore, different instructions may need differently sized encoding (for instance, the implicit `this` parameter should not be passed to static methods), so substituting a generic invoke instruction with a specific one, when the kind of method is finally discovered, may be not so easy.

Other information that are not know are, of course, the return type and the types of formal parameters of the method we want to invoke (and even the *number* of parameters if variable arity parameters, added in Java 5, are considered). After all, avoiding to fix these information prematurely is what makes polymorphic bytecode interesting! The lack of these information is problematic because two arbitrary types can be compatible at source level but be rather different at binary level. If a method expects a `double` and we call it with an `int`, then the compiler will silently add a conversion instruction (`i2d`). More importantly, a temporary[15] of type `double` requires twice the memory space of a temporary of type `int`: this poses the problem of how to allocate the temporaries on the local stack. Furthermore, we cannot predict how much the local stack of a method can grow, something that the current JVMs require to know for security reasons.

The list of problems that we have just discussed is surely incomplete; yet, it should be enough to make it clear that supporting polymorphic bytecode at the JVM level requires a great amount of changes across the architecture of the JVMs *unless* one accepts to verify all constraints of a class when it has to be verified. While we think that implementation complexity is something that should not interfere too much with design decisions, it is a fact that embracing polymorphic bytecode would put *a lot* of burden on JVM implementors while, on the other hand, it would ease the task of compiler writers. For this reason, we chose to begin our study with a conservative approach where all constraints of a class are verified together; in this case, the involved types are known upfront and the code can be treated in the "traditional" way.

From a design point of view, it is not easy to match the behaviour of standard JVMs and there is certainly a spectrum of choices to be explored. Delaying too much the verification of constraints is a double-edged sword: taking this delay to the extreme, that is, waiting until the very last moment to verify the constraints would make the constraints useless: why using constraints if we could directly check each instruction before its first execution?

Leaving the extremes aside, it seems quite problematic to decide when constraints have to be verified. Matching the standard JVMs behaviour would be nice but, as we have seen, it is no picnic.

[15] A compiler generated local variable used to store an intermediate result.

7 RELATED AND FURTHER WORK

Dynamic linking for Java has already been described [6, 13], also in more abstract models covering both the Java and .NET behaviours [7, 8]. How assemblies are resolved, loaded and used in .NET has been modeled in [2]. Of course, modelling standard dynamic linking, these models do not consider the possibility of having type variables inside the bytecode.

Some recent work [3, 4] has introduced the notion of *flexible dynamic linking* in .NET, where type variables are contained in binaries exactly as it happens in polymorphic bytecode [1].

In our approach binaries are equipped with type constraints which drive the process of substituting variables, while [3] is not concerned in *how* substitution are chosen, but rather in *when* they can be chosen and applied maintaining type-safety. Furthermore, the non-deterministic model in [3] allows type variables to appear in field declarations and method signatures as well.

We designed the dynamic linking process as an incremental version of the inter-checking algorithm described in [1], trying to reflect the linking phases and timing from the JVM specification. These design choices led to a deterministic model where each concern (loading, verification and so on) is nicely isolated from the others.

One drawback of our choice is that we need to resolve references earlier than standard JVMs; unfortunately, delaying the resolution of references gives rise to many issues, as we discussed. Our conservative approach exploits polymorphic bytecode to make the linking of Java like languages more flexible without losing the guarantees of statically typed languages; on the contrary, our approach enforces stricter checks than standard JVMs (where a "used" method can not exist if the corresponding invocation is never executed): we allow the linking of classes together only if their sources could be recompiled together. This is a remarkable result, considering that we do not need to know their sources at all! We can achieve this goal thanks to the fact that a polymorphic binary, differently from a standard binary, depends on its source only.

Nevertheless, we feel that making our approach lazier (but not the laziest) would be an important improvement and it is a subject of further work.

On the implementation side, we also need to support some more features of Java in order to promote the polymorphic bytecode approach. In particular, method overloading and (user defined) exceptions are two features that users expect to be available in any Java-like language and that are challenging to deal with.

Acknowlegments We are grateful to Alex Buckley and Sophia Drossopoulou for having provided an incredible amount of insightful comments and helpful suggestions on the previous workshop version of this paper. We also warmly thank Davide Ancona and Elena Zucca for their advice and feedback.

This work has been partially supported by MIUR EOSDUE - Extensible Object Systems for Dynamic and Unpredictable Environments.

REFERENCES

[1] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2005*. ACM Press, January 2005.

[2] Alex Buckley. A Model of Dynamic Binding in .NET. In *Component Deployment 2005*, November 2005.

[3] Alex Buckley and Sophia Drossopoulou. Flexible dynamic linking. In *6th Intl. Workshop on Formal Techniques for Java Programs*, June 2004.

[4] Alex Buckley, Michelle Murray, Susan Eisenbach, and Sophia Drossopoulou. Flexible bytecode for linking in .NET. *Electronic Notes in Theoretical Computer Science*, 141(1):75–92, 2005.

[5] L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, 1997.

[6] Sophia Drossopoulou. An abstract model of Java dynamic linking and loading. In *Types in Compilation*, pages 53–84, 2000.

[7] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible models for dynamic linking. In Pierpaolo Degano, editor, *ESOP 2003 - European Symposium on Programming 2003*, number 2618 in Lecture Notes in Computer Science, pages 38–53. Springer, April 2003.

[8] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. A flexible model for dynamic linking in Java and C#. *Theoretical Computer Science*, 368(1–2):1–29, December 2006.

[9] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java language specification*. The Java series. Addison-Wesley, third edition, 2005.

[10] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.

[11] Giovanni Lagorio. Dynamic linking of polymorphic bytecode. In *8th Intl. Workshop on Formal Techniques for Java-like Programs*, 2006.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Second edition, 1999.

[13] Z. Qian, Al. Goldberg, and A. Coglio. A formal specification of Java class loading. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2000)*, volume 35(10) of *SIGPLAN Notices*, pages 325–336. ACM Press, 2000.

ABOUT THE AUTHORS

Giovanni Lagorio took a Ph.D. in Computer Science at the University of Genova in May 2004. His research interests are in the area of programming languages; in particular, design and foundations of modular and object-oriented languages and systems. He can be reached at lagorio@disi.unige.it.
See also http://www.disi.unige.it/person/LagorioG/.