Dynamic linking of polymorphic bytecode*

Giovanni Lagorio

DISI - Università di Genova
Via Dodecaneso, 35, 16146 Genova (Italy)
email: lagorio@disi.unige.it

Abstract. 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.

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[1] 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 [8]. 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

* Partially supported by APPSEM II - Thematic network IST-2001-38957, and MIUR EOS - Extensible Object Systems.

[1] This is a simplified view: we are not considering class loaders [10, 11].

references to fields and methods[2]. If an error occurs during resolution, then the exception `IncompatibleClassChangeError` or one of its subclasses, for instance `NoSuchMethodError`, is thrown.

The JVM specification [10] 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.

Keeping references to class members in symbolic form inside binaries, as opposed to fixing object layouts at compile time[3], greatly enhances the possibility of reusing binaries in diverse binary environments.

However, binaries could be even more reusable if some of these symbolic references were not fixed at compile time: in standard Java, compilation binaries depend on both their corresponding sources and the compilation environment they are compiled in[4]. This latter dependency poses some unnecessary restrictions on which execution environments can be used to run the code.

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 [2, 3], combining polymorphic bytecode with dynamic linking allows 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. Hence, in this paper we analyze how the JVM specification could be modified in order to make it run polymorphic bytecode natively, and give a model trying to describe as close as possible how a modified JVM could be implemented.

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; making our approach lazier is subject of further work.

Section 2 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 3 defines runtime expressions. Section 4 describes execution; for lack of space we had to omit some technical

[2] Constructors are considered special methods, named `<init>`, at binary level.
[3] As it happens, for instance, in languages as C/C++.
[4] We recall some more details in Section 2.

details, so this is just an overview. Interested readers can find the auxiliary definitions and the missing rewriting rules in the Appendix. Finally, Section 5 discusses related work and concludes.

2 Binary environments

Figure 1 defines binary environments. A binary environment $\mathcal{B}$ is a sequence of binary fragments where each fragment defines a differently named class[5].

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

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

Fig. 1. Binary environments.

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.

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 [9]. 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

[5] 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.

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 declaring no fields or methods, or the declaration of a class `c`, which contains a superclass name $\mathtt{c}'$, 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 `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 to have to 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 $\mathsf{e}^{\mathsf{b}}_{\mathtt{mono}} =$ `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 as well.

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:

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

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:

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

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[6]". 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[7].

The compilation of cast expressions presents another issue to take care of: consider the source expression e^{s} of type `t` and the expression: $\mathsf{e}^{\mathsf{s}}_{\mathtt{cast}} = (\mathtt{c})\mathsf{e}^{\mathsf{s}}$. This cast is correct whenever `t` and `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 `t` and `c` is unknown, then the polymorphic cast expression $\ll \mathtt{c}, \mathtt{t} \gg \mathsf{e}^{\mathsf{b}}$ can be used. When polymorphic bytecode is *instantiated*, that expression is replaced by e^{b} in binary environments where `t` is more specific than `c`, and by a standard cast $(\mathtt{c})\mathsf{e}^{\mathsf{b}}$ 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 `t` is a subtype of $\mathtt{t}'$
- $\phi(\mathtt{t}, \mathtt{f}, \mathtt{t}')$ — type `t` provides a field named `f` of type $\mathtt{t}'$
- $\mu(\mathtt{t}, \mathtt{m}, \bar{\mathtt{t}}, (\mathtt{t}', \bar{\mathtt{t}}'))$ — type `t` provides a method named `m`, applicable to argument types $\bar{\mathtt{t}}$, with parameter types $\bar{\mathtt{t}}'$ and return type $\mathtt{t}'$ (this type of constraint and the following one need to consider both the formal and the actual parameter types to produce standard bytecode out of polymorphic one).
- $\kappa(\mathtt{c}, \bar{\mathtt{t}}, \bar{\mathtt{t}}')$ — class `c` provides a constructor applicable to argument types $\bar{\mathtt{t}}$, with parameter types $\bar{\mathtt{t}}'$
- $\mathtt{c} \sim \mathtt{t}$ — class `c` and type `t` are comparable.

These are the constraints given in [1], with the exception of constraints "$\exists\,\mathtt{c}$", with the informal meaning "class `c` must exist". Indeed, these existential constraints are only needed to make compositional compilation equivalent to standard global compilation. In a JVM we do not need to require the existence of

[6] The variable β is not used in any constraint and can assume any value.

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

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.

3 Runtime expressions

Figure 2 shows runtime expressions; except for `verifyCls` and `bootstrap`, which are peculiar of our approach and are described, respectively, below and in the next section, they are standard: values v, field accesses, method invocations, instance creations, cast expressions and exceptions ϵ.

$$
\begin{array}{rcl}
\mathsf{e}^{\mathsf{r}} & ::= & v \mid \mathsf{e}^{\mathsf{r}}[\mathsf{c.f}\ \mathsf{c}'] \mid \mathsf{e}^{\mathsf{r}}_0[\mathsf{c.m}(\bar{\mathsf{c}})\mathsf{c}'](\mathsf{e}^{\mathsf{r}}_1, \ldots, \mathsf{e}^{\mathsf{r}}_n) \mid \mathtt{new}\ [\mathsf{c}\ \bar{\mathsf{c}}](\overline{\mathsf{e}^{\mathsf{r}}}) \mid \\
 & & (\mathsf{c})\mathsf{e}^{\mathsf{r}} \mid \epsilon \mid \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}) \mid \mathtt{bootstrap}(\bar{\gamma}, \mathsf{e}^{\mathsf{b}}) \\
\overline{\mathsf{e}^{\mathsf{r}}} & ::= & \mathsf{e}^{\mathsf{r}}_1, \ldots, \mathsf{e}^{\mathsf{r}}_n \\
v & ::= & \mathtt{new}\ \mathsf{c}(\bar{v}) \\
\bar{v} & ::= & v_1, \ldots, v_n \\
\epsilon & ::= & \mathtt{NoClassDefFoundError} \mid \mathtt{ClassCircularityError} \mid \mathtt{VerifyError} \mid \\
 & & \mathtt{ClassCastException}
\end{array}
$$

Fig. 2. Syntax of runtime expressions.

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, method invocations and instance creations are annotated like their binary counterpart, but in this case annotations contain no type variables because polymorphic code is verified and *instantiated* before it is executed. That is, type constraints are checked and, when verification succeeds, type variables are replaced by class names found in the execution environment and polymorphic casts replaced as previously described.

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 wrapped around 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 $\mathsf{e}^{\mathsf{r}}_1 = \mathtt{new}\ \mathsf{c}(\ldots)$ is rewritten into $\mathsf{e}^{\mathsf{r}}_2 = \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}_1)$, that can be read as "verify class `c` first, then go on with the execution of $\mathsf{e}^{\mathsf{r}}_1$".

Rewrite rules (Section 4) 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.

4 Execution

Execution, modeled in small step style, has the form: $\mathcal{B}^{\mathsf{L}}_1, \mathcal{B}^{\mathsf{V}}_1, \mathsf{e}^{\mathsf{r}}_1 \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \mathcal{B}^{\mathsf{V}}_2, \mathsf{e}^{\mathsf{r}}_2$ 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[8] in $\mathcal{B}$, and $\mathcal{B}^{\mathsf{L}}_2$ is always equal to or greater than $\mathcal{B}^{\mathsf{L}}_1$.
- $\mathcal{B}^{\mathsf{V}}_1$ and $\mathcal{B}^{\mathsf{V}}_2$ contain the *verified* classes; that is, they contain monomorphic code and no type constraints. $\mathcal{B}^{\mathsf{V}}_2$ is always equal or greater than $\mathcal{B}^{\mathsf{V}}_1$, and each class they contain is also contained in $\mathcal{B}^{\mathsf{L}}_1$ (but in polymorphic form).
- $\mathsf{e}^{\mathsf{r}}_1$ 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 $\mathcal{B}^{\mathsf{V}}$, 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:

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

The binary expression e^{b} corresponds to the code of the `main` method and $\bar{\gamma}$ contains its type constraints. This `bootstrap` expression is either rewritten into a monomorphic expression (see Figure 6 in the Appendix), 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.

Constraint verification is modeled by the execution of verification *actions* $\mathcal{A}$. 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 make the constraints hold.

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 $\mathcal{B}^{\mathsf{L}}_1, \mathcal{A}_1 \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}_2, \mathcal{A}_2$, where:

$$\begin{aligned}\mathcal{A} ::= {} & \texttt{load}(\mathsf{c}, \mathcal{A}) \mid \texttt{verify}(\bar{\gamma}, \mathcal{A}) \mid \texttt{verifyEither}(\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3) \mid \\ & \texttt{match}(\bar{\mathsf{t}}, \bar{\mathsf{c}}, \mathcal{A}) \mid \sigma \mid \epsilon\end{aligned}$$

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

[8] 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.

– load $\mathtt{c}$, then execute $\mathcal{A}$;
– verify $\bar{\gamma}$, then execute $\mathcal{A}$;
– verify either $\mathcal{A}_1$ or $\mathcal{A}_2$, then execute $\mathcal{A}_3$;
– produce a substitution σ matching $\bar{\mathtt{t}}$ with $\bar{\mathtt{c}}$, then execute $\sigma(\mathcal{A})$ – note that this is the standard application of a substitution except when σ is applied to another σ' (inside $\mathcal{A}$): 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.

$$\frac{}{\mathcal{B}^{\mathtt{L}}, \mathtt{load}(\mathtt{c}, \mathcal{A}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathtt{L}}, \mathcal{A}} \quad \mathtt{c} \in def(\mathcal{B}^{\mathtt{L}})$$

$$\frac{}{\mathcal{B}^{\mathtt{L}}, \mathtt{load}(\mathtt{c}, \mathcal{A}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathtt{L}}, \mathtt{NoClassDefFoundError}} \quad \mathtt{c} \notin def(\mathcal{B})$$

$$\frac{}{\mathcal{B}^{\mathtt{L}}_1, \mathtt{load}(\mathtt{c}, \mathcal{A}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathtt{L}}_1, \mathtt{ClassCircularityError}} \quad \begin{array}{l} \mathtt{c} \in def(\mathcal{B} \setminus \mathcal{B}^{\mathtt{L}}_1) \\ \mathcal{B}^{\mathtt{L}}_2 = \mathcal{B}^{\mathtt{L}}_1\, \mathcal{B}(\mathtt{c}) \\ isInsideACycle_{\mathcal{B}^{\mathtt{L}}_2}(\mathtt{c}) \end{array}$$

$$\frac{}{\mathcal{B}^{\mathtt{L}}_1, \mathtt{load}(\mathtt{c}, \mathcal{A}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathtt{L}}_2, \mathcal{A}} \quad \begin{array}{l} \mathtt{c} \in def(\mathcal{B} \setminus \mathcal{B}^{\mathtt{L}}_1) \\ \mathcal{B}^{\mathtt{L}}_2 = \mathcal{B}^{\mathtt{L}}_1\, \mathcal{B}(\mathtt{c}) \\ \neg isInsideACycle_{\mathcal{B}^{\mathtt{L}}_2}(\mathtt{c}) \end{array}$$

Fig. 3. Rewrite rules for loading classes.

As an example, let us consider the rewrite rules for action `load`, shown in Figure 3 (the rewrite rules for the other actions can be found in the Appendix). In an environment where class $\mathtt{c}$ has already been loaded, action $\mathtt{load}(\mathtt{c}, \mathcal{A})$ just vanishes to let the execution continue with $\mathcal{A}$ (first rule). If the requested class cannot be found or its loading would introduce a cycle in the type hierarchy, then the corresponding exception is thrown (second and third rules). Finally, if everything is fine then the verification continues with $\mathcal{A}$ in a new environment $\mathcal{B}^{\mathtt{L}}_2$ where the binary $\mathtt{b}$, loaded from $\mathcal{B}$, has been added to $\mathcal{B}^{\mathtt{L}}_1$. Note that we check that loading a class does not create cycles in the type hierachy, whereas we do not check overriding rules, exactly as it happens in standard JVMs.

In [1] method overloading and field hiding are not modeled, so we resolve type constraints without taking these two features into account[9]. That is, when we search for a method named $\mathtt{m}$, invoked with n arguments, we end the lookup procedure at the first $\mathtt{m}$ accepting n arguments. Analogously, when we search for a field $\mathtt{f}$ we end the lookup procedure at the first field named $\mathtt{f}$. However,

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

we do not need to forbid the presence of overloaded methods or hidden fields to obtain soundness[10], so we do not check their presence when loading a class.

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

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

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

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

$$\frac{\mathcal{B}_1^{\mathsf{L}}, \mathtt{verify}(\bar{\gamma}, \emptyset) \rightsquigarrow_{\mathcal{B}}^{+} \mathcal{B}_2^{\mathsf{L}}, \sigma}{\mathcal{B}_1^{\mathsf{L}}, \mathcal{B}_1^{\mathsf{V}}, \mathtt{verifyCls}(\mathsf{c}, \mathsf{e}^{\mathsf{r}}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}_2^{\mathsf{L}}, \mathcal{B}_2^{\mathsf{V}}, \mathsf{e}^{\mathsf{r}}} \quad \begin{array}{l} \mathtt{readyTBV}(\mathsf{c}, \mathcal{B}_1^{\mathsf{L}}, \mathcal{B}_1^{\mathsf{V}}) \\ (\mathsf{cd}^{\mathsf{b}}, \bar{\gamma}) = \mathcal{B}_1^{\mathsf{L}}(\mathsf{c}) \\ \mathtt{wellFormedAndCompliant}(\bar{\gamma}, \mathsf{cd}^{\mathsf{b}}) \\ \Delta = \mathit{type}(\mathcal{B}_2^{\mathsf{L}}) \\ \mathcal{B}_2^{\mathsf{V}} = \mathcal{B}_1^{\mathsf{V}}\,(\mathbf{I}_{\Delta}^{\sigma}(\mathsf{cd}^{\mathsf{b}}), \Lambda) \end{array}$$

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

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

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

Fig. 4. Rewrite rules for verifying and linking classes.

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 at the bottom of the figure: `readyTBV` and `wellFormedAndCompliant`.

[10] At the moment this is a conjecture: full proofs are work in progress.

The former expresses that a class is "ready To Be Verified" when it 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.

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 to check 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 $\bar{\gamma} \vdash \mathsf{cd}^{\mathsf{b}} \diamond$, 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 [4], this corresponds to the *intra-checking* of cd^{b}, while *inter-checking* happens incrementally when executing `match` actions (triggered by verification).

Compilers could infer the most general type constraints for a given source [1] but, at the JVM level, 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.

Back to the fifth rule: if all the above conditions are met and $\bar{\gamma}$ are successfully verified 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 $\mathcal{B}_2^{\mathsf{V}}$, where the monomorphic class[11] obtained *instantiating* cd^{b} using the substitution σ has been added. The instantiation function $\mathbf{I}_{\Delta}^{\sigma}(\mathsf{cd}^{\mathsf{b}})$, which we have only informally described, has been given in [1].

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

All other rules for verifying type constraints can be found in the Appendix, along with the rules describing normal execution, abnormal execution (that is, exception throwing) and standard closures.

5 Related and further work

Dynamic linking for Java has already been described [5, 11], also in more abstract models covering both the Java and .NET behaviours [6, 7]. Of course, modelling standard dynamic linking, these models do not consider the possibility of having type variables inside the bytecode.

Some recent work [2, 3] 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 [2] is not concerned in *how* substitution

[11] Actually, the pair consisting of the monomorphic class and an empty constraint sequence.

are chosen, but rather in *when* they can be chosen and applied maintaining type-safety. Furthermore, the non-deterministic model in [2] 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.

We expect the execution we have modeled to be sound, but full formal proofs are work in progress.

Future work includes a lazier approach in substituting variables and a prototype implementation to experiment with. On the implementation side, we 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.

Acknowledgements We are grateful to all FTfJP reviewers for their feedback and suggestions. Alex Buckley and Sophia Drossopoulou have provided an incredible amount of insightful comments and helpful suggestions for enhancing the presentation of our model. We warmly thank also Elena Zucca and Davide Ancona for their advice and feedback.

References

1. D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2005*. ACM Press, January 2005.
2. A. Buckley and S. Drossopoulou. Flexible Dynamic Linking. In *6th Intl. Workshop on Formal Techniques for Java Programs 2004*, June 2004.
3. A. Buckley, M. Murray, S. Eisenbach, and S. Drossopoulou. Flexible bytecode for linking in .NET. *Electr. Notes Theor. Comput. Sci*, 141(1):75–92, 2005.
4. L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, 1997.
5. S. Drossopoulou. An abstract model of Java dynamic linking and loading. In *Types in Compilation*, pages 53–84, 2000.
6. S. Drossopoulou, G. Lagorio, and S. Eisenbach. Flexible models for dynamic linking. In P. Degano, editor, *ESOP 2003 - European Symposium on Programming 2003*, pages 38–53, April 2003.
7. S. Drossopoulou, G. Lagorio, and S. Eisenbach. A flexible model for dynamic linking in Java and C#. *Theoretical Computer Science*, 2006. To appear.
8. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification*. The Java series. Addison-Wesley, third edition, 2005.
9. A. Igarashi, B. C. Pierce, and P. 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.

10. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Second edition, 1999.
11. Z. Qian, A. 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, October 2000.

A Auxiliary definitions and remaining rules

$$
\begin{array}{l}
\mathit{declaredClass}(\texttt{class c extends c}' \ \{\ \overline{\mathsf{fd}^{\mathsf{b}}}\ \overline{\mathsf{md}^{\mathsf{b}}}\ \}) = \mathsf{c} \\
\mathit{super}(\texttt{class c extends c}' \ \{\ \overline{\mathsf{fd}^{\mathsf{b}}}\ \overline{\mathsf{md}^{\mathsf{b}}}\ \}) = \mathsf{c}' \\
\mathit{fields}(\texttt{class c extends c}' \ \{\ \overline{\mathsf{fd}^{\mathsf{b}}}\ \overline{\mathsf{md}^{\mathsf{b}}}\ \}) = \overline{\mathsf{fd}^{\mathsf{b}}} \\
\mathit{methods}(\texttt{class c extends c}' \ \{\ \overline{\mathsf{fd}^{\mathsf{b}}}\ \overline{\mathsf{md}^{\mathsf{b}}}\ \}) = \overline{\mathsf{md}^{\mathsf{b}}} \\
\mathit{classDeclaration}(\ (\mathsf{cd}^{\mathsf{b}}, \bar{\gamma})\ ) = \mathsf{cd}^{\mathsf{b}} \\
\mathit{constraints}(\ (\mathsf{cd}^{\mathsf{b}}, \bar{\gamma})\ ) = \bar{\gamma}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{b} \in \mathcal{B} \text{ if } \mathcal{B} = \mathsf{b}_1 \ldots \mathsf{b}_n \text{ and } \exists i : \mathsf{b}_i = \mathsf{b} \\
\mathit{def}(\mathcal{B}) = \bigcup_{\mathsf{b} \in \mathcal{B}} \mathit{declaredClass}(\mathit{classDeclaration}(\mathsf{b})) \\
\mathcal{B}(\mathsf{c}) = \mathsf{b}_i \text{ if } \mathcal{B} = \mathsf{b}_1 \ldots \mathsf{b}_n \text{ and } \mathit{declaredClass}(\mathit{classDeclaration}(\mathsf{b}_i)) \\
\mathcal{B}_1 \leq \mathcal{B}_2 \iff \mathit{def}(\mathcal{B}_1) \supseteq \mathit{def}(\mathcal{B}_2) \text{ and } \forall \mathsf{c} \in \mathit{def}(\mathcal{B}_2) : \mathcal{B}_1(\mathsf{c}) = \mathcal{B}_2(\mathsf{c})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fd}^{\mathsf{b}} \in \overline{\mathsf{fd}^{\mathsf{b}}} \text{ if } \overline{\mathsf{fd}^{\mathsf{b}}} = \mathsf{fd}_1^{\mathsf{b}} \ldots \mathsf{fd}_n^{\mathsf{b}} \text{ and } \exists i : \mathsf{fd}_i^{\mathsf{b}} = \mathsf{fd}^{\mathsf{b}} \\
\mathsf{md}^{\mathsf{b}} \in \overline{\mathsf{md}^{\mathsf{b}}} \text{ if } \overline{\mathsf{md}^{\mathsf{b}}} = \mathsf{md}^{\mathsf{b}}{}_1 \ldots \mathsf{md}^{\mathsf{b}}{}_n \text{ and } \exists i : \mathsf{md}^{\mathsf{b}}{}_i = \mathsf{md}^{\mathsf{b}}
\end{array}
$$

$$
\begin{array}{l}
\mathit{super}_{\mathcal{B}}(\mathsf{c}) = \mathit{super}(\mathit{classDeclaration}(\mathcal{B}(\mathsf{c}))) \\
\mathit{fields}_{\mathcal{B}}(\mathsf{c}) = \mathit{fields}(\mathit{classDeclaration}(\mathcal{B}(\mathsf{c}))) \\
\mathit{methods}_{\mathcal{B}}(\mathsf{c}) = \mathit{methods}(\mathit{classDeclaration}(\mathcal{B}(\mathsf{c})))
\end{array}
$$

$$
\begin{array}{l}
\mathit{subtype}_{\mathcal{B}}(\mathsf{c}, \mathsf{c}') = (\mathsf{c} = \mathsf{c}') \text{ or } \exists n \geq 0 : \ \mathit{super}_{\mathcal{B}}(\mathsf{c}) = \mathsf{c}_1 \ \wedge \ \ldots \ \wedge \ \mathit{super}_{\mathcal{B}}(\mathsf{c}_n) = \mathsf{c}' \\
\mathit{isInsideACycle}_{\mathcal{B}}(\mathsf{c}_0) = \exists n \geq 0 : \ \mathit{super}_{\mathcal{B}}(\mathsf{c}_0) = \mathsf{c}_1 \ \wedge \ \ldots \ \wedge \ \mathit{super}_{\mathcal{B}}(\mathsf{c}_n) = \mathsf{c}_0 \\
\mathit{noCycles}(\mathcal{B}) = \forall \mathsf{c} \in \mathit{def}(\mathcal{B}) : \neg \mathit{isInsideACycle}_{\mathcal{B}}(\mathsf{c})
\end{array}
$$

$$
\begin{array}{l}
\mathit{allFields}_{\mathcal{B}}(\texttt{Object}) = \Lambda \\
\mathit{allFields}_{\mathcal{B}}(\mathsf{c}) = \mathit{allFields}_{\mathcal{B}}(\mathit{super}_{\mathcal{B}}(\mathsf{c}))\, \mathit{fields}_{\mathcal{B}}(\mathsf{c})
\end{array}
$$

$$
\mathit{indexOfField}_{\mathcal{B}}(\mathsf{c}, \mathsf{f}, \mathsf{c}') = i \text{ if } \begin{cases} \mathit{allFields}_{\mathcal{B}}(\mathsf{c}') = \mathsf{fd}_1^{\mathsf{b}} \ldots \mathsf{fd}_n^{\mathsf{b}} \\ \mathsf{fd}_i^{\mathsf{b}} = \mathsf{c}''\ \mathsf{f} \\ \nexists j > i \ : \ \mathsf{fd}_j^{\mathsf{b}} = \mathsf{c}''\ \mathsf{f} \end{cases}
$$

$$
\mathit{method}_{\mathcal{B}}(\mathsf{c}, \mathsf{c}_0, \mathsf{m}, \mathsf{c}_1 \ldots \mathsf{c}_n) =
\begin{cases}
\mathsf{md}^{\mathsf{b}} & \text{if } \mathsf{md}^{\mathsf{b}} = \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1,\ \ldots,\ \mathsf{c}_n\ \mathsf{x}_n)\ \{\ \texttt{return}\ \mathsf{e}^{\mathsf{b}}\texttt{;}\ \} \\
 & \quad \text{and } \mathsf{md}^{\mathsf{b}} \in \mathit{methods}_{\mathcal{B}}(\mathsf{c}) \\
\mathit{method}_{\mathcal{B}}(\mathit{super}_{\mathcal{B}}(\mathsf{c}), \mathsf{c}_0, \mathsf{m}, \mathsf{c}_1 \ldots \mathsf{c}_n) & \text{otherwise}
\end{cases}
$$

Fig. 5. Auxiliary functions and shortcuts.

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

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

$$\frac{}{\Lambda, \Lambda, \texttt{bootstrap}(\bar{\gamma}, \mathsf{e}^{\mathsf{b}}) \rightsquigarrow_{\mathcal{B}} \Lambda, \Lambda, \texttt{VerifyError}} \quad \neg \mathit{wellFormed}(\bar{\gamma}) \vee \bar{\gamma}, \emptyset \not\vdash \mathsf{e}^{\mathsf{b}} : \_$$

Fig. 6. Rewrite rules for bootstrapping the system.

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

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

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

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

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

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

Fig. 7. Compliance of code and constraints.

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

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

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

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

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

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

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

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

$$\frac{}{\mathcal{B}^{\mathtt{L}}, \mathtt{match}(\mathtt{c}, \mathtt{c}, \mathcal{A}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathtt{L}}, \mathcal{A}} \qquad \frac{}{\mathcal{B}^{\mathtt{L}}, \mathtt{match}(\mathtt{c}, \mathtt{c}', \mathcal{A}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathtt{L}}, \mathtt{VerifyError}} \quad \mathtt{c} \neq \mathtt{c}'$$

$$\frac{}{\mathcal{B}^{\mathtt{L}}, \mathtt{match}(\alpha, \mathtt{c}, \mathcal{A}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathtt{L}}, \mathcal{A}[\alpha \mapsto \mathtt{c}]}$$

Fig. 8. Rewrite rules for verifying constraints.

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathtt{verify}(\Lambda, \mathcal{A}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{A}}$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathtt{verify}(\gamma\bar{\gamma}, \mathcal{A}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathtt{verify}(\gamma, \mathtt{verify}(\bar{\gamma}, \mathcal{A}))} \qquad \bar{\gamma} \neq \Lambda$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathtt{match}(\Lambda, \Lambda, \mathcal{A}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{A}}$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathtt{match}(\mathsf{c}\bar{\mathsf{c}}, \mathsf{t}\bar{\mathsf{t}}, \mathcal{A}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathtt{match}(\mathsf{c}, \mathsf{t}, \mathtt{match}(\bar{\mathsf{c}}, \bar{\mathsf{t}}, \mathcal{A}))} \qquad \bar{\mathsf{c}} \neq \Lambda$$

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

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

$$\frac{}{\mathcal{B}^{\mathsf{L}} \vdash \mathsf{c}_1 \leq \mathsf{c}_2} \qquad \begin{array}{l} \mathsf{c}_1 \neq \mathsf{c}_2 \\ \mathtt{Object} \notin \{\mathsf{c}_1, \mathsf{c}_2\} \end{array} \qquad \frac{}{\mathcal{B}^{\mathsf{L}} \vdash \phi(\mathsf{c}, \mathsf{f}, \_)} \qquad \_\ \mathsf{f} \notin fields_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c})$$

$$\frac{}{\mathcal{B}^{\mathsf{L}} \vdash \mu(\mathsf{c}, \mathsf{m}, \_, (\_, \_))} \qquad \_\ \mathsf{m}(\_) \notin methods_{\mathcal{B}^{\mathsf{L}}}(\mathsf{c})$$

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

Fig. 9. Propagation and error rules for verifying constraints.

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathcal{B}^{\mathsf{V}}, \texttt{new [c } \bar{\texttt{c}}\texttt{]}(v_1, \ldots, v_n) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{B}^{\mathsf{V}}, \texttt{verifyCls}(\texttt{c}, \texttt{new [c } \bar{\texttt{c}}\texttt{]}(v_1, \ldots, v_n))} \quad \texttt{c} \notin \mathit{def}(\mathcal{B}^{\mathsf{V}})$$

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

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathcal{B}^{\mathsf{V}}, \texttt{new c}(v_1 \ldots v_n)[\texttt{c}'.\texttt{f c}''] \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{B}^{\mathsf{V}}, v_i} \quad \mathit{indexOfField}_{\mathcal{B}^{\mathsf{V}}}(\texttt{c}', \texttt{f}, \texttt{c}'') = i$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathcal{B}^{\mathsf{V}}, v_0[\texttt{c.m}(\bar{\texttt{c}})\texttt{c}'](v_1, \ldots, v_n) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{B}^{\mathsf{V}}, \texttt{e}^{\texttt{b}}[v_0/\texttt{this}, v_1/\texttt{x}_1, \ldots, v_n/x_n]} \quad \begin{array}{l} v_0 = \texttt{new c}_0(\bar{\texttt{c}}_0) \\ \mathit{method}_{\mathcal{B}^{\mathsf{V}}}(\texttt{c}_0, \texttt{c}', \texttt{m}, \bar{\texttt{c}}) = \\ \quad \texttt{c}'\ \texttt{m}(\texttt{c}_1\ \texttt{x}_1, \ \ldots, \ \texttt{c}_n\ \texttt{x}_n)\ \{\ \texttt{return e}^{\texttt{b}}\texttt{;}\ \} \end{array}$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathcal{B}^{\mathsf{V}}, (\texttt{c})\texttt{new c}'(\bar{v}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{B}^{\mathsf{V}}, \texttt{new c}'(\bar{v})} \quad \mathit{subtype}_{\mathcal{B}^{\mathsf{L}}}(\texttt{c}', \texttt{c})$$

$$\frac{}{\mathcal{B}^{\mathsf{L}}, \mathcal{B}^{\mathsf{V}}, (\texttt{c})\texttt{new c}'(\bar{v}) \rightsquigarrow_{\mathcal{B}} \mathcal{B}^{\mathsf{L}}, \mathcal{B}^{\mathsf{V}}, \texttt{ClassCastException}} \quad \neg \mathit{subtype}_{\mathcal{B}^{\mathsf{L}}}(\texttt{c}', \texttt{c})$$

Fig. 10. Rewrite rules for execution.

$$[\cdot]^{Exp} ::= [\cdot][\texttt{c.f c}'] \mid [\cdot][\texttt{c.m}(\bar{\texttt{c}})\texttt{c}'](\overline{\texttt{e}^{\texttt{r}}}) \mid v[\texttt{c.m}(\bar{\texttt{c}})\texttt{c}'](\bar{v}, [\cdot], \overline{\texttt{e}^{\texttt{r}}}) \mid \texttt{new [c } \bar{\texttt{c}}\texttt{]}(\bar{v}, [\cdot], \overline{\texttt{e}^{\texttt{r}}}) \mid (\texttt{c})[\cdot]$$

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

Fig. 11. Contextual closure for regular and abnormal execution.

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

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

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

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

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

Fig. 12. Contextual closure for execution of `verifyEither`.