JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 11
Special issue: OOPS track at SAC 2004, Nicosia/Cyprus

A Case-Study in Encoding Configuration Languages: Multiple Class Loaders

Sonia Fagorzi and **Elena Zucca**, DISI, Università di Genova, Italy

The contribution of the paper is twofold. First, we define a toy language, called MCL, which provides a very abstract view of the mechanism of dynamic class loading with multiple loaders as in Java. The aim is to study this feature in isolation, allowing a better understanding; moreover, this also shows a stratified approach, which, differently from the Java approach based on reflection, distinguishes between the language at the user level and the configuration language. This approach is less flexible but allows to statically check type safety, hence provides an intermediate solution between the rigid approach based only on the class path and that which allows loaders to depend on execution of user applications, which can be intricate and error-prone.
The second contribution is related to a recent stream of work aiming at defining simple and powerful calculi providing a common foundation for systems supporting dynamic reconfiguration. We use MCL as an extended case-study, by defining an encoding in one of these kernel calculi, and prove the correctness of the translation.

1 INTRODUCTION

Modern programming environments support and will support more and more in the future forms of *dynamic reconfiguration*, in the sense that steps in the execution of an application can be interleaved with reconfiguration steps, that is, steps of combination and manipulation of the code fragments which compose the application itself. However, systems supporting dynamic reconfiguration still lack clear foundations; in particular, existing module/fragment calculi [6, 5, 16, 12] are based on a *static* view of module manipulation, in which open code fragments can be flexibly combined together, but all module operators must be performed once for all *before* starting execution of a program.

Hence, the definition of analogous kernel calculi providing a simple unifying model for various dynamic reconfiguration mechanisms is an important open problem. In a recent stream of papers [1, 3, 2], we have provided first attempts in this direction.

Here we define an encoding in the $CMS^{\ell,\ell-}$ calculus (our most recent proposal [3], which incorporates and improves features from previous work) of a toy language, called MCL, which provides an abstract view of the mechanism of dynamic class loading with multiple loaders as in Java [11, 10, 13].

The aim is twofold. On the one hand, we show an extended case-study of application of the calculus for modeling existing loading and linking policies. On the other

Cite this article as follows: Sonia Fagorzi, Elena Zucca: "A Case-Study in Encoding Configuration Languages: Multiple Class Loaders", in *Journal of Object Technology*, vol. 3, no. 11, December 2004, Special issue: OOPS track at SAC 2004, Nicosia/Cyprus, pp. 31–53, http://www.jot.fm/issues/issue_2004_12/article2

hand, differently from existing informal and formal descriptions of Java class loading [11, 10, 13], which are rather heavy in what their aim is to cover all language-specific aspects, we provide a simple model allowing to understand the key mechanism without to go in too many details.

MCL is a two level language with a user language for defining executable applications and a configuration language for defining class loaders to be used in applications. Both levels have the same syntax[1], a simple functional Java subset, where classes only contain static methods and there is no inheritance. Indeed, the only features we are interested in modeling here are the following: class loading is dynamically triggered whenever the first reference to a class name is encountered (in our simple language, an instance creation or static method invocation), and the actual class which is loaded is not uniquely determined by the class name, since different class loaders can be used in the same application. Classes at the configuration level are called *loaders*, and must have some special methods which are used to load (user) classes. Loaders can manipulate user classes as first-class values, but not conversely, differently from what happens in the Java reflective approach.

In MCL there are two execution levels corresponding to the two language levels. Execution of the user application can trigger, and depends on, execution of the configuration program, but not conversely. Both levels are formalized by a small-step operational semantics. The model for the user level is a model of (a simple subset of) Java execution focusing on class loading, which abstracts from all orthogonal features. Indeed, in addition to considering a very simple language, we also do not model bytecode verification, nor, as already said, the Java approach based on reflection where loaders are just special user classes (subclasses of the class ClassLoader).

One motivation is that, as done in previous work on other Java features (e.g., inheritance and late binding in [9], checked exceptions in [4]), we want to study just one aspect in isolation.

Moreover, we want to show an alternative stratified approach, distinguishing between the language at the user level and the configuration language. Loaders can still be flexibly programmed (by a "superuser" which could be the user himself), but user program execution cannot affect their behaviour. This approach is less flexible w.r.t. the Java behavior, where both aspects are part of the same language, hence loaders might change during execution. On the other hand, this approach looks simpler, safer and less error-prone, so we believe it can be a compromise between a rigid approach based only on the class path and a total freedom in writing user-defined loaders. The approach also looks closer (even though much more flexible) to what happens in .NET, where there is no support for user-defined loaders but some freedom in finding code to be associated to assembly names can be achieved by using an XML configuration file.

[1] This is both for simplicity of presentation and for better mimicking the behavior of loaders in Java. Of course it is just a choice among others: the important point here is to have a separate language for defining loaders.

Formally, stratification allows to statically type-check a user program after executing a configuration program in a given context.

The rest of the paper is organized as follows. In Section 2 we formally define MCL (syntax and reduction rules modeling execution for the two language levels). In Section 3 we give a type system for the two language levels. In Section 4 we briefly recall the calculus $CMS^{\ell,\ell-}$ introduced in [2]. In Section 5 we define a translation from MCL into $CMS^{\ell,\ell-}$ and show that this translation preserves the semantics. Finally in the Conclusion we summarize the contribution of the paper and outline further work.

2 A TWO-LEVEL LANGUAGE WITH MULTIPLE CLASS LOADERS

Notations
We denote by $A \xrightarrow{fin} B$ the set of the partial functions f from A to B with finite domain, written $\mathsf{dom}(f)$; the image of f is written $\mathsf{img}(f)$; $\emptyset$ denotes the function with empty domain, $a_1 : b_1, \ldots, a_n : b_n$ the function which returns b_i on a_i and is undefined otherwise. Finite sequences $a_1, \ldots, a_n$ are also written $a_i^{i \in 1..n}$ for short.

Syntax

MCL is a two level language with a user level for defining executable applications and a configuration level for defining class loaders. Both levels have the same syntax, which is given in Fig.1.

$p \in \mathsf{Prg}$	::=	$cd_1 \ldots cd_n$	program
$cd \in \mathsf{CDec}$	::=	$\mathsf{class}\ c\ \{mds\}$	class declaration
mds	::=	$md_1 \ldots md_n$	
md	::=	$\mathsf{static}\ t\ m\ (t_1\ x_1, \ldots, t_n\ x_n)\ \{e\}$	method declaration
$t \in \mathsf{Type}$	::=	$\mathsf{int} \mid \mathsf{bool} \mid \ldots \mid c \mid \mathsf{cname} \mid \mathsf{cdec} \mid \mathsf{class}$	type
$e \in \mathsf{Exp}$	::=		expression
		x	variable
	|	$n \mid \mathsf{true} \mid \mathsf{false} \mid e_1 + e_2 \mid \ldots$	operator of primitive type
	|	$\mathsf{if}\ (e)\ \{e_1\}\ \mathsf{else}\{e_2\}$	conditional
	|	$\mathsf{new}\ c()$	instance creation
	|	$c.m(e_i^{i \in n})$	method invocation
	|	c	class name

Figure 1: MCL syntax

Metavariables c, m and x range over primitive sets of *class names*, *method names*, and *variables*, respectively. We denote by CName the set of class names.

A program consists of a sequence of class declarations (we assume no multiple declarations for the same class name).

A class declaration consists of a class name and a sequence of static method declarations, each one specifying the return type, name, parameters and body of the method, which is an expression. We assume no multiple declarations for the same method name (that is, no overloading), and that parameter names in a method declaration are distinct. Note that the fact that loaders are classes which must have some special methods will be later imposed by the type system.

Types are either primitive types, or class names, or *metatypes* which can only be used at the configuration level to manipulate class names, class declarations and classes of the user level (this constraint will be imposed by the type system in next section).

There are the following kinds of expressions: variable (parameter), application of an operator of primitive type, conditional, instance creation, (static) method invocation, constant of metatype cname.

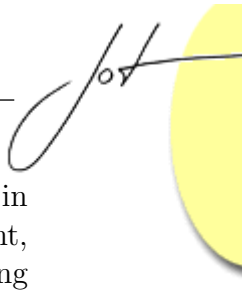
We only consider static methods because this is enough to illustrate dynamic loading; however, we keep instance creation new $c()$ to have a constant of type c in the language.

Java class loading

Before presenting the formal semantics of MCL, we briefly recall how the Java class loading mechanism works. Our description is mainly extracted from [13], to which, together with [11, 10], we refer for more details.

Class loading is the process of obtaining a representation of a class declaration, called a *class file*, and installing the representation within the Java Virtual Machine (JVM). A class file is typically produced by compiling a Java class declaration, and contains essentially the same information except that method bodies are compiled to *bytecode*, that is, an assembly-like language which can be executed by JVM. Bytecode instructions use (class) names to refer to classes. The JVM allows lazy, dynamic loading of classes and a form of name space separation using loaders. Indeed, loading of a class happens only when the first reference to a class name is encountered during program execution. Moreover, distinct loaded classes may have the same name, and within an executing JVM each loaded class is identified by its name plus the class loader that has loaded it (that is, the *defining* loader of the class, see below). Java supports *reflection*, that is, classes have run-time representations in an executing JVM, which are objects, in turn instances of the system class named Class, and they are called *class objects*.

Loaders are objects of subclasses of class ClassLoader. This class contains a defineClass method (final, that is, which cannot be overridden in subclasses) which takes a class file (in the form of a byte array) as argument and returns a newly created class object, unless the class file has an invalid format, in which case an exception is thrown. If the defineClass method is invoked on a loader and a class object is returned, then the loader is called the *defining loader* of the resulting class.



Class ClassLoader also contains a loadClass method, which may be overridden in subclasses. This method takes a class name (in the form of a string) as argument, and returns a class object (or throws the error NoClassDefFoundError). By overriding this method, users can implement arbitrary loading policies. Typically the code will fetch a class file in some way (e.g., from a local directory or remote site), and then invoke defineClass with the resulting class file as argument; moreover, loading can be delegated by calling loadClass upon another loader. If the loadClass method is invoked by the JVM on a loader and a class object is returned, then the loader is called an *initiating loader* of the resulting class. When a class needs to be resolved within an executing class, the defining loader of the executing class is used as initiating loader for the class (name) to be resolved.

Finally, the JVM internally maintains a *loaded class cache* [13, 11, 10], which keeps track of loading requests. The loaded class cache is needed since the JVM cannot trust any user-defined loadClass method to consistently return the same class for a given name [10]. Hence, the loaded class cache guarantees that the JVM never invokes the loadClass method with the same name on the same class loader twice (hence a class can be identified by its name and defining loader).

Semantics of the configuration level

In MCL, differently from Java, there are two execution levels corresponding to the two language levels. Execution of the user application can trigger, and depends on, execution of a configuration program, but not conversely. Both levels are formalized by a small-step operational semantics. Note that this also implies that programmers who are only allowed to write user level code cannot write their own loaders.

To avoid confusion, we call *loaders* classes of the configuration level (expected to have some special methods, as will be enforced by the type system). Moreover, even though they range over the same set CName, for clarity we use ℓ and c as metavariables for loaders and user class names, respectively.

In Fig.2 we show the shape of the reduction relation for the configuration level.

$\xrightarrow[p,\mathcal{U}]{}$	$\subseteq$	CnfState $\times$ (CnfState $\cup$ {err(c) $\mid$ $c \in$ CName})	(config. level) reduction
$\mathcal{U}$			universe
$\sigma \in$ CnfState	::=	(e, L)	(config. level) state
$L \in$ LCCache	=	CName $\times$ CName $\xrightarrow{fin}$ CName $\times$ CDec	loaded class cache
e	::=	$\ldots \mid \mathsf{ret}_{\ell,c}(e) \mid cd \mid (\ell, cd)$	run-time expression
$v \in$ CnfVal	::=	$n \mid$ true $\mid$ false $\mid \ldots \mid$ new $\ell() \mid$ $c \mid cd \mid (\ell, cd)$	(config. level) value

Figure 2: MCL reduction relation for the configuration level

At the configuration level, there is no dynamic loading of loaders, hence the semantics we give is very close to standard execution models for Java-like languages

which do not deal with JVM specific features (see, e.g., [9, 8]), hence evaluation of an expression takes place in a fixed program context, modeled by the index p in the reduction relation.

However, we also parameterize reduction w.r.t. a *universe* $\mathcal{U}$ which models in an abstract way "the external world", that is, file systems, URLs on the web, and so on. The intuitive idea is that evaluation of an expression in the configuration language can depend on $\mathcal{U}$ (for instance, the body of a loadClass method could attempt to get a class with a certain name from a given directory). Since we do not want to bother with details here, the nature of the universe is left unspecified, and we just introduce a primitive method cdec getClassDec(cname) which loaders must provide, whose effect, depending on $\mathcal{U}$, is to either return a class definition with the required name or an error.

(Configuration) states consist of two components: the expression currently evaluated, and a mapping from *loading requests* (pairs consisting of a loader and a user class name) into pairs consisting of a (defining) loader and a class declaration. Indeed, by this mapping we explicitly model the loaded class cache maintained by the JVM previously described.

In our execution model, the loaded class cache is updated each time a call ℓ.loadClass(c) terminates successfully returning a class value (see rule (load-class-ret) in Fig.3). To this end, we enrich expressions by adding run-time expressions of the form $\mathsf{ret}_{\ell,c}\,(e)$ which model intermediate steps in the execution of a call ℓ.loadClass(c). We also add as run-time expressions (user) class declarations, which can be obtained by invoking the primitive method cdec getClassDec(cname), see rule (get-cdec) in Fig.3, and (user) classes, which can be obtained by invoking the primitive method class defineClass(cdec), see rule (define-class) in Fig.3.

The reduction relation maps a state into another state or into an error, which models the situation in which a loader does not find a class (corresponding to NoClassDefFoundError in Java), or the class has not the required name.

Values are values of primitive types, instance creations, constants of metatypes, that is, (user) class names and declarations, and classes.

In Fig.3 we give the reduction rules for the configuration level.

Rules ($\mathcal{C}[\ ]$) and ($\mathcal{C}[\ ]$-err) are the usual contextual closure, where evaluation contexts are those standard for evaluation from left to right of arguments of primitive operators (we show sum as an example), conditional, (static) method invocation and intermediate loadClass execution. The subsequent three rules are those standard for sum and the conditional. Note that instance creation is a value (since we consider a functional subset of Java, as in [9]), hence there is no corresponding rule. Rule (meth) models the method invocation step, which takes place when the arguments have been evaluated. The class declaration associated to the receiver is extracted from the program, and the current expression to be evaluated is updated to the body of the invoked method where formal parameters have been replaced by arguments. The functions cdef and mbody are defined by:

$$\mathcal{C}[\,] ::= \Box \mid \Box + e \mid v + \Box \mid \ldots \mid \mathsf{if}\,(\Box)\,\{e_1\}\,\mathsf{else}\{e_2\} \mid \ell.m(v_1, \ldots, v_{i-1}, \Box, e_i, \ldots e_n) \mid \mathsf{ret}_{\ell,c}\,(\Box) \qquad \text{evaluation context}$$

$$(\mathcal{C}[\,]) \ \frac{(e, L) \xrightarrow[p,\mathcal{U}]{} (e', L')}{(\mathcal{C}[e], L) \xrightarrow[p,\mathcal{U}]{} (\mathcal{C}[e'], L')} \qquad (\mathcal{C}[\,]\text{-err}) \ \frac{(e, L) \xrightarrow[p,\mathcal{U}]{} \mathsf{err}(c)}{(\mathcal{C}[e], L) \xrightarrow[p,\mathcal{U}]{} \mathsf{err}(c)}$$

$$(\text{sum}) \ \frac{}{(v_1 + v_2, L) \xrightarrow[p,\mathcal{U}]{} (v_1 +^{\mathbb{Z}} v_2, L)}$$

$$(\text{if-t}) \ \frac{}{(\mathsf{if}\,(\mathtt{true})\,\{e_1\}\,\mathsf{else}\{e_2\}, L) \xrightarrow[p,\mathcal{U}]{} (e_1, L)}$$

$$(\text{if-f}) \ \frac{}{(\mathsf{if}\,(\mathtt{false})\,\{e_1\}\,\mathsf{else}\{e_2\}, L) \xrightarrow[p,\mathcal{U}]{} (e_2, L)}$$

$$(\text{meth}) \ \frac{}{(\ell.m(v_i{}^{i\in 1..n}), L) \xrightarrow[p,\mathcal{U}]{} (e\{x_i : v_i{}^{i\in 1..n}\}, L)} \quad \begin{array}{l} m \not\equiv \mathsf{defineClass}, \mathsf{getClassDec}, \mathsf{loadClass} \\ \mathsf{cdef}(p, \ell) = cd \\ \mathsf{mbody}(cd, m) = (x_1 \ldots x_n, e) \end{array}$$

$$(\text{define-class}) \ \frac{}{(\ell.\mathsf{defineClass}(cd), L) \xrightarrow[p,\mathcal{U}]{} ((\ell, cd), L)}$$

$$(\text{get-cdec}) \ \frac{}{(\ell.\mathsf{getClassDec}(c), L) \xrightarrow[p,\mathcal{U}]{} \sigma^{\mathcal{U},c}} \quad \sigma^{\mathcal{U},c} ::= (\mathsf{class}\ c\ \{mds\}, L) \mid \mathsf{err}(c)$$

$$(\text{load-class}) \ \frac{}{(\ell.\mathsf{loadClass}(c), L) \xrightarrow[p,\mathcal{U}]{} (\mathsf{ret}_{\ell,c}\,(e)\{x : c\}, L)} \quad \begin{array}{l} (\ell, c) \notin \mathsf{dom}(L) \\ \mathsf{cdef}(p, \ell) = cd \\ \mathsf{mbody}(cd, \mathsf{loadClass}) = (x, e) \end{array}$$

$$(\text{load-class-ret}) \ \frac{}{(\mathsf{ret}_{\ell,c}\,(\ell_d, cd), L) \xrightarrow[p,\mathcal{U}]{} ((\ell_d, cd), L\{(\ell, c) : (\ell_d, cd)\})}$$

$$(\text{loaded-class-cache}) \ \frac{}{(\ell.\mathsf{loadClass}(c), L) \xrightarrow[p,\mathcal{U}]{} ((\ell_d, cd), L)} \quad L(\ell, c) = (\ell_d, cd)$$

Figure 3: MCL reduction rules for the configuration level

$\mathsf{cdef}(cd_1 \ldots cd_n, c) = cd_i$
if $cd_i = \mathsf{class}\ c\ \{mds\}$ for some $i \in 1..n$,
undefined otherwise
$\mathsf{mbody}(cd, m) = (x_1 \ldots x_n, e)$
if $cd = \mathsf{class}\ c\ \{md_1 \ldots md_k\}$, $md_i = \mathsf{static}\ t\ m\ (t_1\ x_1, \ldots, t_n\ x_n)\,\{e\}$ for some $i \in 1..k$,
undefined otherwise

Invocations of special methods defineClass, getClassDec and loadClass need to be handled in a special way, as shown by the following rules which have no counterpart at the user level. Rule (define-class) handles an invocation of the primitive method defineClass, which simply returns the class value consisting of the loader on which the method has been invoked and the class declaration passed as argument. This rule mimics the behaviour of the defineClass method in Java, which constructs a

Class object from a byte array (representing a class file). However, here we do not want to bother about low-level details such as the difference between the source class declaration, the class file and its byte array representation. Hence, we take an abstract view in which the argument is directly a class declaration and a class value is just a pair consisting of a loader and a class declaration. In particular, we do not consider the Java exception which is thrown in case the class file has an invalid format. Rule (get-cdec) handles an invocation of the primitive method getClassDec, which returns, depending on the universe $\mathcal{U}$, either a class declaration with the name passed as argument, or an error. Rule (load-class) handles an invocation of the method loadClass in the case the loading request has not been encountered before. This invocation is handled as a standard invocation apart that the body is enclosed into a $\mathsf{ret}_{\ell,c}\,(\_)$ context. This is used in the following rule (load-class-ret) to update the loaded class cache when the body execution terminates. Rule (loaded-class-cache) handles an invocation of the method loadClass in the case the loading request has already been encountered. In this case, the class stored in the loaded class cache is simply returned.

Semantics of the user level

In Fig.4 we show the shape of the reduction relation for the user level.

$\xrightarrow[p,\mathcal{U}]{} \subseteq \mathsf{UsrState} \times (\mathsf{UsrState} \cup \{\mathsf{err}(c) \mid c \in \mathsf{CName}\})$	(user level) reduction
$\sigma \in \mathsf{UsrState} ::= (\mathcal{S};\ L;\ e)$	(user level) state
$\mathcal{S} ::= \ell_1, \ldots, \ell_n$	loader stack
$e ::= \ldots \mid \mathsf{ret}\,(e)$	run-time expression
$v \in \mathsf{UsrVal} ::= n \mid \mathsf{true} \mid \mathsf{false} \mid \ \ldots \ \mid \mathsf{new}\ c()$	(user level) value

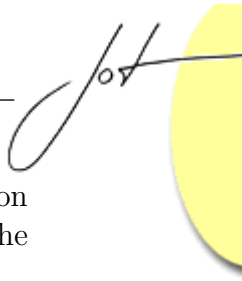
Figure 4: MCL reduction relation for the user level

User level states model snapshots of the execution of a user application and consist of three components: a stack of loaders, where the top of the stack corresponds to the defining loader of the code currently in execution, the loaded class cache, which is updated by configuration level execution triggered by loading requests, and the expression currently in execution. The reduction relation maps a state into another state or into a "class not found" error propagated from the configuration level execution.

Expressions are enriched by adding run-time expressions of the form $\mathsf{ret}\,(e)$ which model intermediate steps in the execution of a method call.

A value is either a value of a primitive type, or an instance creation.

In Java, execution of an application starts by invoking a class name, say c. The effect is that the initial loader, say ℓ (typically the system loader), initiates the loading of c, an initial class is loaded with defining loader ℓ_d (possibly coinciding with ℓ), and the body of the main method of this class, say e, is executed, with current loader



ℓ_d. Here for simplicity we do not model main methods and assume that execution starts from an initial state where the stack of loaders contains one element, and the loaded class cache is empty.

In Fig.5 we give the reduction rules for the user level.

$$\mathcal{C}[\,] \;::=\; \Box \mid \Box + e \mid v + \Box \mid \ldots \mid \mathsf{if}\,(\Box)\,\{e_1\}\,\mathsf{else}\{e_2\} \mid \ell.m(v_1,\ldots,v_{i-1},\Box,e_i,\ldots e_n) \mid \mathsf{ret}\,(\Box) \qquad \text{evaluation context}$$

$$(\mathcal{C}[\,])\ \frac{(\mathcal{S};\ L;\ e) \xrightarrow[p,\mathcal{U}]{} (\mathcal{S}';\ L';\ e')}{(\mathcal{S};\ L;\ \mathcal{C}[e]) \xrightarrow[p,\mathcal{U}]{} (\mathcal{S}';\ L';\ \mathcal{C}[e'])} \qquad (\mathcal{C}[\,])\text{-err}\ \frac{(\mathcal{S};\ L;\ e) \xrightarrow[p,\mathcal{U}]{} \mathsf{err}(c)}{(\mathcal{S};\ L;\ \mathcal{C}[e]) \xrightarrow[p,\mathcal{U}]{} \mathsf{err}(c)}$$

$$(\mathrm{load})\ \frac{(\ell.\mathsf{loadClass}(c), L) \xrightarrow[p,\mathcal{U}]{\star} (v, L')}{(\ell,\mathcal{S};\ L;\ e) \xrightarrow[p,\mathcal{U}]{} (\ell,\mathcal{S};\ L';\ e)} \quad \begin{array}{l} e \;::=\; \mathsf{new}\ c() \mid c.m({v_i}^{i\in 1..n}) \\ (\ell,c) \notin \mathsf{dom}(L) \end{array}$$

$$(\text{load-err})\ \frac{(\ell.\mathsf{loadClass}(c), L) \xrightarrow[p,\mathcal{U}]{\star} \mathsf{err}(c)}{(\ell,\mathcal{S};\ L;\ e) \xrightarrow[p,\mathcal{U}]{} \mathsf{err}(c)} \quad \begin{array}{l} e \;::=\; \mathsf{new}\ c() \mid c.m({v_i}^{i\in 1..n}) \\ (\ell,c) \notin \mathsf{dom}(L) \end{array}$$

$$(\mathrm{meth})\ \frac{}{\left(\ell,\mathcal{S};\ L;\ c.m({v_i}^{i\in 1..n})\right) \xrightarrow[p,\mathcal{U}]{} (\ell_d,\ell,\mathcal{S};\ L;\ e')} \quad \begin{array}{l} L(\ell,c) = (\ell_d, cd) \\ \mathsf{mbody}(cd,m) = ({x_i}^{i\in 1..n}, e) \\ e' = \mathsf{ret}\left(e\{x_i : {v_i}^{i\in 1..n}\}\right) \end{array}$$

$$(\mathrm{ret})\ \frac{}{(\ell,\mathcal{S};\ L;\ \mathsf{ret}\,(v)) \xrightarrow[p,\mathcal{U}]{} (\mathcal{S};\ L;\ v)}$$

Figure 5: MCL reduction rules for the user level

We omit standard rules for primitive operators and conditional, which are analogous to those for the configuration level. Rule (load) is new w.r.t. the configuration level, and models the situation in which a class name c is encountered (in our simple language this only happens in instance creation and static method invocation) with current loader ℓ and the loading request (ℓ, c) has not been considered yet. In this case, execution at the configuration level is triggered by invoking the method loadClass on ℓ with argument c. We denote by $\xrightarrow[p,\mathcal{U}]{\star}$ the reflexive and transitive closure of the reduction relation at the configuration level. Execution at the configuration level will update the loaded class cache L, adding in particular the association from (ℓ, c) into the corresponding defining loader and class declaration. Note, however, that nothing prevents a loader from deciding to load other classes in response to some loading request, hence in general we need to pass the whole loaded class cache between the two levels. Rule (load-err) deals with the case in which the execution at the configuration level results into an error. This can happen if the loadClass method of the current loader ℓ does not succeed in loading the required class name. In this case, the rule simply propagates this error to the user level execution. Rule (meth) is analogous to that for the configuration level. However, here the rule can only be applied when, besides arguments having been evaluated, the receiver's class has been already loaded; in this case, the loaded class cache is used to get the defining loader ℓ_d and the class declaration associated to the loading

request. Note also that the ret operator is applied to the method body. Moreover, the defining loader ℓ_d of the class to which the invoked method belongs becomes the new current loader (that is, it is put on top of the stack of loaders). Finally, note that at the user level class declarations are found (after the initial load) in the loaded class cache, whereas at the configuration level, since we do not model dynamic loading of loaders, they are extracted from the current program context, which is an index in the reduction relation. We could have equivalently assumed to have all loader classes as default entries in the loaded class cache; however, we preferred this approach since it makes more clear the stratification in two levels. Rule (ret) models the end of the evaluation of a method body, when we obtain a value. In this case, a pop operation is performed on the stack of loaders, so that the current loader comes back to that at the time of the method invocation.

3 TYPE SYSTEM

The fact that MCL is a stratified language allows one to statically check type soundness of a user application w.r.t. a type environment obtained by running a configuration program in a given context.

Configuration level

The type system for the configuration level is a standard type system for a Java-like language (see, e.g., [9, 8]). The typing judgments and typing rules for states, programs, class and method declarations are shown in Fig.6.

$p, \mathcal{U} \vdash (e, L)$	(e, L) well-formed configuration state w.r.t. $p, \mathcal{U}$
$\vdash p$	p well-formed program
$\Gamma \vdash cd$	cd well-formed class declaration $w.r.t.$ Γ
$\Gamma \vdash md$	md well-formed method declaration $w.r.t.$ Γ
$\Gamma : \mathsf{CName} \xrightarrow{fin} \mathsf{CType}$	class type environment
$ct \in \mathsf{CType} ::= mh_1 \ldots mh_n$	class type
$mh ::= t\ m(t_1 \ldots t_n)$	method header

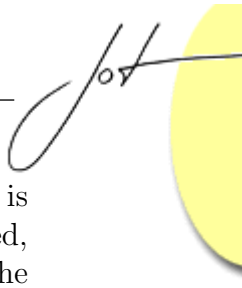
$$(\text{state})\ \frac{\vdash p \quad \mathsf{type}(p); \emptyset \vdash e : t \quad \lambda_{p,\mathcal{U}} \vdash L}{p, \mathcal{U} \vdash (e, L)}$$

$$(\text{prg})\ \frac{\mathsf{type}(p) \vdash cd_i \qquad \forall i \in 1..n}{\vdash p} \quad p = cd_1 \ldots cd_n$$

$$(\text{cdec})\ \frac{\Gamma \vdash md_i \qquad \forall i \in 1..n}{\Gamma \vdash \mathsf{class}\ \ell\ \{md_1 \ldots md_n\}} \quad \exists i\ s.t.\ \mathsf{type}(md_i) = \mathsf{class}\,\mathsf{loadClass}(\mathsf{cname})$$

$$(\text{mdec})\ \frac{\Gamma; x_1 : t_1, \ldots, x_n : t_n \vdash e : t}{\Gamma \vdash \mathsf{static}\ t\ m\ \ (t_1\ x_1, \ldots, t_n\ x_n)\,\{e\}}$$

Figure 6: MCL type system for the configuration level (1)



A configuration level state consisting of an expression and a loaded class cache is well-formed w.r.t. a given program and a universe $\mathcal{U}$ if the program is well-formed, the expression is well-formed w.r.t. the class type environment extracted from the program and the empty local type environment and the loaded class cache agrees with the *loading environment associated to* p,$\mathcal{U}$ (see in the type system for the user level).

A class type environment is a mapping from class names into class types, which are sequences of method headers (consisting of return type, name and parameter types of a method).

The class type environment $\mathsf{type}(p)$ extracted from p is defined as follows:

$$
\begin{array}{lcl}
\mathsf{type}(cd_1 \ldots cd_n) & = & \mathsf{type}(cd_1) \ldots \mathsf{type}(cd_n) \\
\mathsf{type}(\mathsf{class}\ c\ \{mds\}) & = & c : \mathsf{type}(mds) \\
\mathsf{type}(\mathsf{class}\ \ell\ \{mds\}) & = & \ell : (\mathsf{type}(mds), \mathsf{class}\,\mathsf{defineClass}(\mathsf{cdec}), \\
 & & \mathsf{cdec}\,\mathsf{getClassDec}(\mathsf{cname}) \\
\mathsf{type}(md_1 \ldots md_n) & = & \mathsf{type}(md_1) \ldots \mathsf{type}(md_n) \\
\mathsf{type}(\mathsf{static}\ t\ m\ \ (t_1\ x_1, \ldots, t_n\ x_n)\ \{e\}) & = & t\ m(t_1 \ldots t_n)
\end{array}
$$

Note that each loader has, besides the declared methods, the primitive methods $\mathsf{class}\,\mathsf{defineClass}(\mathsf{cdec})$ and $\mathsf{cdec}\,\mathsf{getClassDec}(\mathsf{cname})$.

A program is well-formed if each class declaration in it is well-formed w.r.t. the class type environment extracted from the program.

A class declaration is well-formed w.r.t. a class type environment Γ if each method declaration in the class is well-formed w.r.t. Γ and, moreover, the class declares a $\mathsf{class}\,\mathsf{loadClass}(\mathsf{cname})$ method.

A method declaration is well-formed w.r.t. a class type environment Γ if the body is well-formed w.r.t. Γ and the local type environment which associates to each parameter its type.

The typing judgment and typing rules for expressions are shown in Fig.7.

A local type environment is a mapping from variables into types.

Rule (var), (n), (`true`), (`false`), (sum), (if), (new) and (meth) are standard. The function mtype is defined in the following way:

$$\mathsf{mtype}(mh_1 \ldots mh_n, m) = mh_i \quad \text{if} \quad mh_i = t\ m(t_1 \ldots t_n).$$

It extracts the method header with a certain name in a class declaration, if any. In rule (c), a user class name has the metatype cname. Finally, rule (ret) states that the $\mathsf{ret}_{\ell,c}\,(\_)$ operator applied to an expression does not change its type.

User level

The type system for the user level is based on the following key features. First, a user state can be statically type-checked only having the information on how

$\Gamma; \Pi \vdash e : t$ $\quad$ e well-formed expression with type t *w.r.t.* Γ and Π

$\Pi : \mathsf{Var} \xrightarrow{fin} \mathsf{Type}$ $\quad$ local type environment

$$(\text{var})\ \frac{}{\Gamma; \Pi \vdash x : \Pi(x)} \qquad (n)\ \frac{}{\Gamma; \Pi \vdash n : \mathsf{int}} \qquad (\mathsf{true})\ \frac{}{\Gamma; \Pi \vdash \mathtt{true} : \mathsf{bool}}$$

$$(\mathsf{false})\ \frac{}{\Gamma; \Pi \vdash \mathtt{false} : \mathsf{bool}} \qquad (\text{sum})\ \frac{\Gamma; \Pi \vdash e_1 : \mathsf{int} \quad \Gamma; \Pi \vdash e_2 : \mathsf{int}}{\Gamma; \Pi \vdash e_1 + e_2 : \mathsf{int}}$$

$$(\text{if})\ \frac{\Gamma; \Pi \vdash e : \mathsf{bool} \quad \Gamma; \Pi \vdash e_1 : t \quad \Gamma; \Pi \vdash e_2 : t}{\Gamma; \Pi \vdash \mathsf{if}\,(e)\,\{e_1\}\,\mathsf{else}\{e_2\} : t} \qquad (\text{new})\ \frac{}{\Gamma; \Pi \vdash \mathsf{new}\ \ell() : \ell}$$

$$(\text{meth})\ \frac{\Gamma; \Pi \vdash e_i : t_i \quad \forall i \in n}{\Gamma; \Pi \vdash \ell.m(e_1, \ldots e_n) : t}\ \mathsf{mtype}(\Gamma(\ell), m) = t\ m(t_1 \ldots t_n)$$

$$(c)\ \frac{}{\Gamma; \Pi \vdash c : \mathsf{cname}} \qquad (\text{ret})\ \frac{\Gamma; \Pi \vdash e : t}{\Gamma; \Pi \vdash \mathsf{ret}_{\ell,c}\,(e) : t}$$

Figure 7: MCL type system for the configuration level (2)

loading requests would be resolved by executing a configuration program in a given context. This is formally modeled by a *loading environment* which is a mapping that associates to each loading request (pair consisting of an initiating loader and a class name) a class (pair consisting of a defining loader and a class declaration) and a loaded class cache. Moreover, a class name cannot be directly used as type of a user level expression, but it must be *tagged* with its defining loader, which can be obtained via the loading environment from the initiating loader at that point. To this end, typing judgments are parameterized by the current loader (in the case of expressions, by the current loader stack).

The typing judgments and typing rules for states, class declarations environments, class and method declarations are shown in Fig.8.

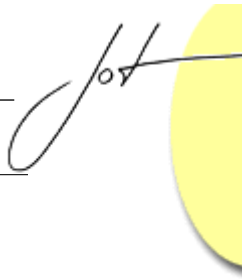
In rule (state) $\lambda_{p,\mathcal{U}}$ denotes the *loading environment associated to* p,$\mathcal{U}$, that is, the loading environment obtained by executing in p and $\mathcal{U}$ all the possible loading requests:

$$\lambda_{p,\mathcal{U}}(\ell, c) = \begin{cases} ((\ell_d, cd), L) & \text{if } (\ell.\mathsf{loadClass}(c), \emptyset) \xrightarrow[p,\mathcal{U}]{\star} ((\ell_d, cd), L) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Defining loaders and *class declarations environments* extracted from a loading environment λ are defined in the following way:

- $\lambda^{\mathsf{CName}}(\ell, c) = \ell_d$ $\quad$ if $\quad$ $\lambda(\ell, c) = ((\ell_d, \_), \_)$
- $\lambda^{\mathsf{CDec}}(\ell, c) = cd$ $\quad$ if $\quad$ $\lambda(\ell, c) = ((\_, cd), \_)$

The class type environment $\mathsf{type}(\mathcal{CD})$ extracted from a class declarations environment $\mathcal{CD}$ is defined analogously to that extracted from a configuration program,



$p,\mathcal{U} \vdash (\mathcal{S};\ L;\ e)$ $(\mathcal{S};\ L;\ e)$ well-formed user state *w.r.t.* $p,\mathcal{U}$

$\Gamma;\mathcal{D} \vdash \mathcal{CD}$ $\mathcal{CD}$ well-formed class declarations environment *w.r.t.* $\Gamma,\mathcal{D}$

$\Gamma;\mathcal{D} \vdash_\ell cd$ cd well-formed class declaration *w.r.t.* $\Gamma,\mathcal{D}$

$\Gamma;\mathcal{D} \vdash_\ell md$ md well-formed method declaration *w.r.t.* $\Gamma,\mathcal{D}$

$\lambda : \mathsf{CName} \times \mathsf{CName} \xrightarrow{fin} (\mathsf{CName} \times \mathsf{CDec}) \times \mathsf{LCCache}$ loading environment

$\Gamma : \mathsf{CName} \times \mathsf{CName} \xrightarrow{fin} \mathsf{CType}$ class type environment

$\mathcal{D} : \mathsf{CName} \times \mathsf{CName} \xrightarrow{fin} \mathsf{CName}$ defining loaders environment

$\mathcal{CD} : \mathsf{CName} \times \mathsf{CName} \xrightarrow{fin} \mathsf{CDec}$ class declarations environment

$$\text{(state)}\ \frac{\Gamma;\mathcal{D} \vdash \lambda^{\mathsf{CDec}}_{p,\mathcal{U}} \qquad \Gamma;\mathcal{D};\emptyset \vdash_{\mathcal{S}} e : \tau \qquad \lambda_{p,\mathcal{U}} \vdash L}{p,\mathcal{U} \vdash (\mathcal{S};\ L;\ e)} \quad \Gamma = \mathsf{type}(\lambda^{\mathsf{CDec}}_{p,\mathcal{U}}), \mathcal{D} = \lambda^{\mathsf{CName}}_{p,\mathcal{U}}$$

$$\text{(cdec-env)}\ \frac{\{\Gamma;\mathcal{D} \vdash_{\ell_d} cd \mid \mathcal{CD}(\ell,c) = cd, \mathcal{D}(\ell,c) = \ell_d\}}{\Gamma;\mathcal{D} \vdash \mathcal{CD}}$$

$$\text{(cdec)}\ \frac{\Gamma;\mathcal{D} \vdash_\ell md_i \qquad \forall i \in 1..n}{\Gamma;\mathcal{D} \vdash_\ell \mathsf{class}\ c\ \{md_1 \ldots md_n\}}$$

$$\text{(mdec)}\ \frac{\Gamma;\mathcal{D};x_1 : \tau_1,\ldots,x_n : \tau_n \vdash_\ell e : \tau}{\Gamma;\mathcal{D} \vdash_\ell \mathsf{static}\ t\ m\ (t_1\ x_1,\ldots,t_n\ x_n)\,\{e\}} \quad \begin{array}{l}\tau_i = \tilde{\mathcal{D}}\,(\ell,t_i)\ \ \forall i \in 1..n \\ (\star)\ \tau = \tilde{\mathcal{D}}\,(\ell,t)\end{array}$$

Figure 8: MCL type system for the user level (1)

except that there are loading requests instead of class names and there are no primitive methods:

$$\mathsf{type}(\mathcal{CD})(\ell,c) = \mathsf{type}(mds) \text{ iff } \mathcal{CD}(\ell,c) = \mathsf{class}\ c\ \{mds\}$$

A user state consisting of a loader stack, a loaded class cache and an expression is well-formed w.r.t. a configuration program p and a universe $\mathcal{U}$ if the following conditions hold: the class declarations environment associated to $p,\mathcal{U}$ is well-formed w.r.t. the class type and defining loaders environments associated to $p,\mathcal{U}$; the expression is well-formed w.r.t. the class type environment extracted fromw.r.t. the class type and defining loaders environments associated to $p,\mathcal{U}$, the empty local type environment and the loader stack (see typing judgment and rules for expressions in the following); the loaded class cache agrees with the loading environment associated to $p,\mathcal{U}$ (see below).

The judgment $\lambda \vdash L$ holds if $\mathrm{L}(\ell,c) = (\ell_d, cd)$ implies $\lambda(\ell,c) = ((\ell_d, cd), \_)$.

A class declarations environment $\mathcal{CD}$ is well formed w.r.t. a class type enviroment Γ and a defining loaders environment $\mathcal{D}$ if each class declaration which could be possibly loaded is well-formed w.r.t. Γ, $\mathcal{D}$ and its defining loader ℓ_d.

A class declaration is well formed w.r.t. a class type environment Γ, a defining loaders environment $\mathcal{D}$ and a loader ℓ if each method declaration in it is well-formed w.r.t. Γ, $\mathcal{D}$ and ℓ.

A method declaration is well-formed w.r.t. the class type environment Γ, the defining loaders environment $\mathcal{D}$ and the current loader ℓ if the body is well-formed w.r.t. Γ, $\mathcal{D}$, the local type environment which associates to each parameter its (tagged) type, obtained through $\tilde{\mathcal{D}}(\ell, \_)$ (see below) and ℓ. Moreover, we have to check that the type of the body corresponds (through $\tilde{\mathcal{D}}(\ell, \_)$) to the one declared in the class.

The typing judgment and typing rules for expressions are shown in Fig.9.

$\Gamma; \mathcal{D}; \Pi \vdash_{\mathcal{S}} e : \tau$ — e well-formed expression with (tagged) type τ *w.r.t.* $\Gamma, \mathcal{D}, \Pi$ and $\mathcal{S}$

$\tau ::= \mathsf{int} \mid \mathsf{bool} \mid \ldots \mid (\ell, c)$ — tagged type

$$(\text{ret})\ \frac{\Gamma; \mathcal{D}; \Pi \vdash_{\mathcal{S}} e : \tau}{\Gamma; \mathcal{D}; \Pi \vdash_{\mathcal{S},\ell} \mathsf{ret}\,(e) : \tau} \qquad (\text{sum-1})\ \frac{\Gamma; \mathcal{D}; \Pi \vdash_{\mathcal{S},\ell} e_1 : \mathsf{int} \quad \Gamma; \mathcal{D}; \Pi \vdash_{\ell} e_2 : \mathsf{int}}{\Gamma; \mathcal{D}; \Pi \vdash_{\mathcal{S},\ell} e_1 + e_2 : \mathsf{int}}\ e_1 \notin \mathsf{UsrVal}$$

$$(\text{sum-2})\ \frac{\Gamma; \mathcal{D}; \Pi \vdash_{\ell} v : \mathsf{int} \quad \Gamma; \mathcal{D}; \Pi \vdash_{\mathcal{S},\ell} e : \mathsf{int}}{\Gamma; \mathcal{D}; \Pi \vdash_{\mathcal{S},\ell} v + e : \mathsf{int}} \qquad (\text{new})\ \frac{}{\Gamma; \mathcal{D}; \Pi \vdash_{\ell} \mathsf{new}\ c() : \tilde{\mathcal{D}}(\ell, c)}$$

$$(\text{meth})\ \frac{\Gamma; \mathcal{D}; \Pi \vdash_{\ell} v_k : \tau_k \quad \forall k \in 1..i-1 \quad \Gamma; \mathcal{D}; \Pi \vdash_{\mathcal{S},\ell} e_i : \tau_i \quad \Gamma; \mathcal{D}; \Pi \vdash_{\ell} e_j : \tau_j \quad \forall j \in i+1..n}{\Gamma; \mathcal{D}; \Pi \vdash_{\mathcal{S},\ell} c.m(v_1, \ldots v_{i-1}, e_i, \ldots e_n) : \tilde{\mathcal{D}}(\ell_d, t)}$$

$e_i \notin \mathsf{UsrVal}$
$\mathcal{D}(\ell, c) = \ell_d$
$\Gamma(\ell_d, c) = ct$
$\mathsf{mtype}(ct, m) = t\ m(t_1\ \ldots\ t_n)$
$(\star)\ \tilde{\mathcal{D}}(\ell_d, t_i) = \tau_i\ \forall i \in 1..n$

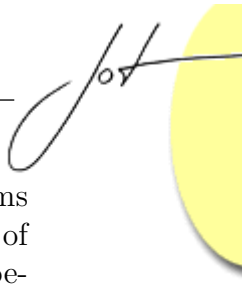
Figure 9: MCL type system for the user level (2)

An expression is well-formed and has a given *tagged type* w.r.t. a class type environment, a defining loaders environment, a local type environment and a loader stack. A tagged type is either a primitive type or a class name tagged with a (defining) loader. Indeed, class names in user code will be associated to different classes depending on the current loader.

In the rules, $\tilde{\mathcal{D}}(\_, \_)$ denotes a function which, given an initiating loader, transforms types into types tagged by the corresponding defining loader by applying $\mathcal{D}$, defined by: $\tilde{\mathcal{D}}(\ell, \mathsf{int}) = \mathsf{int}, \tilde{\mathcal{D}}(\ell, \mathsf{bool}) = \mathsf{bool}, \tilde{\mathcal{D}}(\ell, c) = (\mathcal{D}(\ell, c), c)$.

The role of the loader stack in type-checking expressions is the following. Expressions in user programs (that is, not containing $\mathsf{ret}\,(e)$ subexpressions) can be type-checked by just having the current loader. However, they can reduce to run-time expressions where, each time the evaluation of a method body starts, method body is boxed into a ret operator, and, correspondingly, a new loader is pushed onto the stack. Hence, the current loader must be determined starting from the bottom of the stack and going up to the next (upper) one in $\mathcal{S}$ each time a $\mathsf{ret}\,(e)$ expression is encountered. This is expressed by rule (ret).

Note that a run-time expression obtained by applying reduction rules can contain



(recursively) only one ret (e) subterm, and all previous (in leftmost order) subterms must have been already evaluated. This is expressed by the rules for operators of primitive types, conditional, and method call, where the loader stack is used to type-check only the first argument which is not a value yet, while the other arguments are typed w.r.t. the loader at the bottom of the stack (we only show the two rules (sum-1) and (sum-2) for the sum operator).

Rules for variables, constants of primitive types and conditional are omitted (they are analogous to those for the configuration level, considering a loader stack with only the current loader). The (new) rule states that the type of an instance creation expression new $c()$ contained in code with current loader ℓ is the tagged type (ℓ_d, c) where ℓ_d is the defining loader obtained taking ℓ as initiating loader for c. In rule (meth), arguments are first type-checked, getting their (tagged) types. As already mentioned, the loader stack is used to type-check the first argument which is not a value yet, while the other arguments are type-checked w.r.t. the loader at the bottom of the stack. Then, the defining loader of the receiver is found in the defining loaders environment, the class type environment is used to get the associated class declaration, and the method type is extracted. Argument types must match parameter types tagged by their defining loader (which is obtained by taking ℓ_d, that is, the current loader for the method declaration, as initiating loader). Finally, the type of the method call is the return type of the method tagged by its defining loader as well.

A very important point to mention is about the two side conditions marked by $(\star)$ in rules (mdec) and (meth), respectively. These side conditions, as the reader can see, express *constraints* on the classes which can be loaded. In the JVM, these constraints are checked dynamically by maintaining an internal data structure (*loading constraints* [13, 10]). In earlier versions of the JVM, these constraints were not checked, leading to type violations. This bug was firstly reported by [14]. In our approach, since loading environment cannot be affected by the user program execution, these constraints are statically enforced by the type system, as shown above; even more, they just emerge naturally by adapting the standard Java type system to the case when classes are loaded from a loading environment.

Results

We give now the technical results about the language. In particular, we state the Progress and Subject Reduction properties for the reduction relation at the two levels. These properties are expected for the configuration level, whose type system is a variant of standard type system for Java-like languages. On the contrary, type soundness for the user level is a novelty of our approach, where user programs can be statically type-checked after fixing a loading environment. Note that this also prevents user level reduction to raise err(c) errors.

Proposition 3.1 (Progress)

- *If* $p \vdash (e, L)$ *and* $e \notin \mathsf{CnfVal}$, *then* $(e, L) \xrightarrow[p,\mathcal{U}]{} \sigma$ *with* $\sigma ::= (e', L') \mid \mathsf{err}(c)$.
- *If* $p, \mathcal{U} \vdash (\mathcal{S};\ L;\ e)$ *and* $e \notin \mathsf{UsrVal}$, *then* $(\mathcal{S};\ L;\ e) \xrightarrow[p,\mathcal{U}]{} (\mathcal{S}';\ L';\ e')$.

Proof *Both the two facts are proven by induction on the structure of* e *and case-analysis on the last rule applyed in the derivation of the first judgement.* ∎

Proposition 3.2 (Subject Reduction)

- *If* $p \vdash (e, L)$ *and* $(e, L) \xrightarrow[p,\mathcal{U}]{} (e', L')$, *then* $p \vdash (e', L')$
- *If* $p, \mathcal{U} \vdash (\mathcal{S};\ L;\ e)$ *and* $(\mathcal{S};\ L;\ e) \xrightarrow[p,\mathcal{U}]{} (\mathcal{S}';\ L';\ e')$, *then* $p, \mathcal{U} \vdash (\mathcal{S}';\ L';\ e')$.

Proof *Both the two facts are proven by induction on the derivation of reduction and case-analysis on the last typing rule applyed.* ∎

4 AN OVERVIEW OF THE TARGET CALCULUS

In this section we briefly present (a subset of) the $CMS^{\ell,\ell\text{-}}$ calculus which we will use for encoding MCL. The reader can refer to [2] for other operators, all technical details and an extended discussion on the motivations and the expressive power of the calculus.

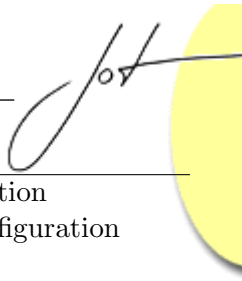
$CMS^{\ell,\ell\text{-}}$ is a module calculus where steps of execution of a module component can be interleaved with reconfiguration steps, that is, reductions at the module level, and execution can partly control precedence between these reconfiguration steps. This is achieved by means of a *low priority link* operator which is only performed when a certain component, which has not been linked yet, is both available and needed for execution to proceed, otherwise precedence is given to the outer operators. In this way, control over precedence between this operator and others can be obtained by appropriately using variables in user's code.

The syntax of the (subset of) the calculus is given in Fig.10.

We assume infinite sets Name of *names* X, Var of *variables* x, and Exp of (core) expressions (the expressions of the underlying language used for defining module components). Indeed, $CMS^{\ell,\ell\text{-}}$ is a parametric and stratified calculus, which can be instantiated over different core calculi. In the following section we will consider a particular instantiation in order to encode MCL.

For our aim here, it is enough to consider a very small subset of $CMS^{\ell,\ell\text{-}}$ which only includes basic configurations and the low priority link operator. However, we also include in our presentation the sum operator, to give the flavour of the calculus and of more powerful configuration mechanisms which could be encoded as well.

A $CMS^{\ell,\ell\text{-}}$ executable configuration is obtained applying operators (here only sum and low-priority link) to a basic executable configuration $< [\iota; o; \rho], e >$ which



$C \in \mathsf{Conf}$	$::=$			executable configuration
		$< [\iota;\, o;\, \rho]\,,\ e >$	$(\mathsf{dom}(\iota) \cap \mathsf{dom}(\rho) = \emptyset)$	executable basic configuration
	$\mid$	$C + M$		sum
	$\mid$	$\mathsf{link}^{-}{}_{\sigma} C$		low priority link
$M \in \mathsf{MExp}$	$::=$			non-executable configuration
	$\mid$	$[\iota;\, o;\, \rho]$	$(\mathsf{dom}(\iota) \cap \mathsf{dom}(\rho) = \emptyset)$	non-executable basic configuration
	$\mid$	$M + M$		sum
$\iota : \mathsf{Var} \xrightarrow{fin} \mathsf{Name}$				input assignment
$o : \mathsf{Name} \xrightarrow{fin} \mathsf{Exp}$				output assignment
$\rho : \mathsf{Var} \xrightarrow{fin} \mathsf{Exp}$				local assignment
$\sigma : \mathsf{Name} \xrightarrow{fin} \mathsf{Name}$				renaming
$e \in \mathsf{Exp}$	$::=$	$x \mid \ldots$		(core) expression
$t \in \mathsf{Type}$	$::=$	$\ldots$		(core) type

Figure 10: CMS^{ℓ,ℓ^-} syntax

intuitively models execution of a core program (expression e) in the context offered by $[\iota;\, o;\, \rho]$. Expression e may contain variables which are either *local*, that is, have an associated definition in ρ, or *deferred*, that is, have no associated definition, but are bound to an *input name* in ι.

Evaluation of e is performed by applying reduction rules at the core level (rule (core)), and replacing local variables by their definitions (rule (var)) until a deferred variable is encountered. In this case rule (var/err) is applied and an error is raised which means "execution is stuck since input component X is needed, and currently available output components are $\mathcal{Y}$". In this case, reconfiguration steps must be performed until X becomes available. In the small subset we consider, a reconfiguration step can be either a linking step, which consists in binding X to a currently available output name Y in o (rule (link^-/basic)), thus making local all deferred variables associated with X, or performing a sum with an external module (rule (sum/basic)), or switching a low-priority link operator which is not applicable with an outer operator (combined effect of remaining rules).

The following examples give the flavour of CMS^{ℓ,ℓ^-} reduction. In

$$\mathsf{link}^{-}{}_{Y:Y}(\mathsf{link}^{-}{}_{X:X} < [x : X, y : Y;\, X : 2;\,]\,,\ x + 1 > + [;\, Y : 2;\,]),$$

since execution needs component X, the $\mathsf{link}^{-}{}_{X:X}$ operator is executed and the configuration reduces to $\mathsf{link}^{-}{}_{Y:Y}(< [y : Y;\, X : 2;\, x : 2]\,,\ x + 1 > + [;\, Y : 2;\,])$.

However, if the execution needs the component Y instead, e.g.,

$$\mathsf{link}^{-}{}_{Y:Y}(\mathsf{link}^{-}{}_{X:X} < [x : X, y : Y;\, X : 2;\,]\,,\ y + 1 > + [;\, Y : 2;\,]),$$

then the $\mathsf{link}^{-}{}_{X:X}$ is not performed, and outer operators are moved inside and performed instead, as shown below.

$$\mathsf{link}^{-}{}_{Y:Y}(\mathsf{link}^{-}{}_{X:X} < [x : X, y : Y;\, X : 2;\,]\,,\ y + 1 > + [;\, Y : 2;\,]) \rightarrow$$
$$\mathsf{link}^{-}{}_{Y:Y}\mathsf{link}^{-}{}_{X:X} < [x : X, y : Y;\, X : 2, Y : 2;\,]\,,\ y + 1 > \rightarrow$$

$$(M\text{-sum})\ \frac{}{M_1 + M_2 \rightarrow [\iota_1, \iota_2;\ o_1, o_2;\ \rho_1, \rho_2]}\quad \begin{array}{l} M_i \equiv [\iota_i;\ o_i;\ \rho_i]\,, i \in \{1,2\} \\ (\mathsf{dom}(\iota_1) \cup \mathsf{dom}(\rho_1)) \cap \mathsf{FV}(M_2) = \emptyset \\ (\mathsf{dom}(\iota_2 \cup \mathsf{dom}(\rho_2)) \cap \mathsf{FV}(M_1) = \emptyset \end{array}$$

$$(\text{core})\ \frac{e \xrightarrow[\text{core}]{} e'}{< [\iota;\ o;\ \rho]\,,\ e > \rightarrow < [\iota;\ o;\ \rho]\,,\ e' >}$$

$$(\text{var})\ \frac{}{< [\iota;\ o;\ \rho]\,,\ \mathcal{E}\,[x] > \rightarrow < [\iota;\ o;\ \rho]\,,\ \mathcal{E}\,[\rho(x)] >}\quad \begin{array}{l} x \in \mathsf{dom}(\rho) \\ \mathcal{E}\,[x] \not\xrightarrow[\text{core}]{} \end{array}$$

$$(\text{var/err})\ \frac{}{< [\iota;\ o;\ \rho]\,,\ \mathcal{E}\,[x] > \rightarrow \mathsf{err}(X, \mathcal{Y})}\quad \begin{array}{l} \iota(x) = X \\ \mathcal{E}\,[x] \not\xrightarrow[\text{core}]{} \\ \mathsf{dom}(o) = \mathcal{Y} \end{array}$$

$$(\text{sum/basic})\ \frac{< [\iota;\ o;\ \rho]\,,\ e > \rightarrow \mathsf{err}(X, \mathcal{Y}) \qquad [\iota_1;\ o_1;\ \rho_1] + [\iota_2;\ o_2;\ \rho_2] \rightarrow [\iota;\ o;\ \rho]}{< [\iota_1;\ o_1;\ \rho_1]\,,\ e > + [\iota_2;\ o_2;\ \rho_2] \rightarrow < [\iota;\ o;\ \rho]\,,\ e >}$$

$$(\mathsf{link}^{\text{-}}\text{/basic})\ \frac{< M,\ e > \rightarrow \mathsf{err}(X, \mathcal{Y})}{\mathsf{link}^{\text{-}}{}_{X:Y} < M,\ e > \rightarrow < \left[\iota_{\setminus L};\ o;\ \rho, x : o(Y)^{x \in L}\right],\ e >}\quad \begin{array}{l} M \equiv [\iota;\ o;\ \rho] \\ Y \in \mathcal{Y} \\ L = \{x \mid \iota(x) = X\} \end{array}$$

$$(\text{sum/}\mathsf{link}^{\text{-}})\ \frac{\mathsf{link}^{\text{-}}{}_{\sigma} C \rightarrow \mathsf{err}(X, \mathcal{Y})}{\mathsf{link}^{\text{-}}{}_{\sigma} C + [\iota;\ o;\ \rho] \rightarrow \mathsf{link}^{\text{-}}{}_{\sigma}(C + [\iota;\ o;\ \rho])}$$

$$(\mathsf{link}^{\text{-}})\ \frac{C \rightarrow \mathsf{err}(X, \mathcal{Y})}{\mathsf{link}^{\text{-}}{}_{\sigma, X:Y} C \rightarrow \mathsf{link}^{\text{-}}{}_{\sigma} \mathsf{link}^{\text{-}}{}_{X:Y} C}\ Y \in \mathcal{Y}$$

$$(\mathsf{link}^{\text{-}}\text{/err})\ \frac{C \rightarrow \mathsf{err}(X, \mathcal{Y})}{\mathsf{link}^{\text{-}}{}_{\sigma} C \rightarrow \mathsf{err}(X, \mathcal{Y})}\ X \notin \mathsf{dom}(\sigma) \vee \sigma(X) \notin \mathcal{Y}$$

Figure 11: $CMS^{\ell,\ell\text{-}}$ reduction rules

$$\mathsf{link}^{\text{-}}{}_{X:X} \mathsf{link}^{\text{-}}{}_{Y:Y} < [x : X, y : Y;\ X : 2, Y : 2;\ ]\,,\ y + 1 > \rightarrow$$
$$\mathsf{link}^{\text{-}}{}_{X:X} < [x : X;\ X : 2, Y : 2;\ y : 2, z : 2]\,,\ y + 1 >$$

In Fig.11 we give the reduction rules (rules for contextual closure are omitted). Arrow $\xrightarrow[\text{core}]{}$ and $\mathcal{E}\,[\,]$ denotes reduction and evaluation contexts at the core level, respectively. In rule (var), $\mathsf{FV}(M)$ denotes the free variables of M, whose definition, omitted here (see [2]), is on top of that for the core level.

5 TRANSLATION

As target language we consider a particular instantiation of $CMS^{\ell,\ell\text{-}}$ (see Fig.12).

We assume that, for each pair consisting of a loader and a class name, there exist a distinguished variable $x(\ell, c)$ and a distinguished name $X(\ell, c)$.

Core expressions include core variables (which encode pairs consisting of a loader and a class name), application of an operator of primitive type, conditional, instance creation, method invocation, class declaration and let-in expression. In the sequel

$$\mathsf{Var} = \{x(\ell, c) \mid \ell, c \in \mathsf{CName}\}, \quad \mathsf{Name} = \{X(\ell, c) \mid \ell, c \in \mathsf{CName}\}$$

$$\begin{array}{rcl} e \in \mathsf{Exp} & ::= & x(\ell,c) \mid n \mid \texttt{true} \mid \texttt{false} \mid e_1 + e_2 \mid \ldots \mid \mathsf{if}\,(e)\,\{e_1\}\,\mathsf{else}\{e_2\} \mid \mathsf{new}\ e() \mid \\ & & e.m(e_1, \ldots e_n) \mid cd \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ v \in \mathsf{Val} & ::= & n \mid \texttt{true} \mid \texttt{false} \mid \mathsf{new}\ cd() \\ \mathcal{E}\,[\,] & ::= & \Box \mid \Box + e \mid v + \Box \mid \ldots \mid \mathsf{if}\,(\Box)\,\{e_1\}\,\mathsf{else}\{e_2\} \mid \mathsf{new}\ \Box() \mid \\ & & \Box.m(e_1, \ldots e_n) \mid cd.m(v_1, \ldots, v_{i-1}, \Box, e_i, \ldots e_n) \mid \mathsf{let}\ x = \Box\ \mathsf{in}\ e \end{array}$$

$$\text{(c-sum)}\ \frac{}{(v_1 + v_2, L) \xrightarrow[p,\mathcal{U}]{} (v_1 +^{\mathbb{Z}} v_2, L)}$$

$$\text{(c-if-t)}\ \frac{}{\mathsf{if}\,(\texttt{true})\,\{e_1\}\,\mathsf{else}\{e_2\} \xrightarrow[\mathsf{core}]{} e_1} \qquad \text{(c-if-f)}\ \frac{}{\mathsf{if}\,(\texttt{false})\,\{e_1\}\,\mathsf{else}\{e_2\} \xrightarrow[\mathsf{core}]{} e_2}$$

$$\text{(c-meth)}\ \frac{}{cd.m(v_i{}^{i\in 1..n}) \xrightarrow[\mathsf{core}]{} e\{x_i : v_i{}^{i\in 1..n}\}} \quad \mathsf{mbody}(cd, m) = (x_1 \ldots x_n, e)$$

$$\text{(c-let)}\ \frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e \xrightarrow[\mathsf{core}]{} e\{x : v\}} \qquad (\mathcal{E}\,[\,])\ \frac{e \xrightarrow[\mathsf{core}]{} e'}{\mathcal{E}\,[e] \xrightarrow[\mathsf{core}]{} \mathcal{E}\,[e']}$$

Figure 12: $CMS^{\ell,\ell\text{-}}$ instantiation

we will abbreviate a let-in construct where the binding variable does not appear in the body of the expression as follows: $e_1; e_2 \stackrel{\Delta}{=} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ if $x \notin \mathsf{FV}(e_2)$. Class declarations are included since in the encoding, whenever a class named c is loaded with initiating loader ℓ, the variable $x(\ell, c)$ is replaced by the actual class declaration, by applying rule (var) of $CMS^{\ell,\ell\text{-}}$.

Values contain, in addition to values of primitive type, the new application to a class declaration. Core evaluation contexts and rules are standard.

The translation is defined in Fig.13. We use the superscripts MCL and $CMS^{\ell,\ell\text{-}}$ to denote syntactic categories of MCL and $CMS^{\ell,\ell\text{-}}$, respectively.

A MCL user state is translated into a $CMS^{\ell,\ell\text{-}}$ configuration; the translation is parameterized on a fixed program p and a fixed universe $\mathcal{U}$.

The configuration consists of a basic configuration to which a low-priority link operator is applied. More precisely:

Input assignment The input assignment keeps track of the loading requests which are not yet in the cache. For each (variable corresponding to) a loading request (ℓ, c) not in L, there is an input component which maps $x(\ell, c)$ into the name $X(\ell_d, c)$ with ℓ_d defining loader for this request, obtained by invoking the `loadClass` method of the loader ℓ, which is the initiating loader for the class c to load. Recall that $\lambda^{\mathsf{CName}}_{p,\mathcal{U}}(\ell, c) = \ell_d$ stands for $(\ell.\mathsf{loadClass}(c), \emptyset) \xrightarrow[p,\mathcal{U}]{\star} ((\ell_d, \_), \_)$.

The *loaded class caches environment* extracted from λ is defined by:

$$\lambda^{\mathsf{LCCache}}(\ell, c) = L \quad \text{if} \quad \lambda(\ell, c) = ((\_, \_), L)$$

$\mathcal{L} : \mathsf{CName} \times \mathsf{CName} \xrightarrow{fin} \mathsf{LCCache}$ loaded class caches environment

$\boxed{\mathsf{T}_{p,\mathcal{U}} : \mathsf{UsrState}^{\mathsf{MCL}} \to \mathsf{Conf}^{CMS^{\ell,\ell\text{-}}}}$

$\mathsf{T}_{p,\mathcal{U}}(\mathcal{S};\ L;\ e) = \mathsf{link}^{\text{-}}{}_{\sigma}{<}\,[\iota;\ o;\ \rho]\,,\ \mathsf{T}_{\lambda^{\mathsf{LCCache}}_{p,\mathcal{U}},\mathcal{S}}(e)\ >$, with:

$$\iota = \left\{x(\ell,c) : X(\ell_d,c) \mid \lambda^{\mathsf{CName}}_{p,\mathcal{U}}(\ell,c) = \ell_d, (\ell,c) \notin L\right\}$$

$$o = \left\{X(\ell_d,c) : \mathsf{T}_{\lambda^{\mathsf{LCCache}}_{p,\mathcal{U}},\ell_d}(cd) \mid \lambda^{\mathsf{CName}}_{p,\mathcal{U}}(\ell,c) = \ell_d \wedge \lambda^{\mathsf{CDec}}_{p,\mathcal{U}}(\ell,c) = cd\right\}$$

$$\rho = \left\{x(\ell,c) : \mathsf{T}_{\lambda^{\mathsf{LCCache}}_{p,\mathcal{U}},\ell_d}(cd) \mid L(\ell,c) = (\ell_d, cd)\right\}$$

$$\sigma = \{X(\ell_d,c) : X(\ell_d,c) \mid X(\ell_d,c) \in \mathsf{cod}(\iota)\}$$

$\boxed{\mathsf{T}_{\mathcal{L},\ell} : \mathsf{CDec}^{\mathsf{MCL}} \to \mathsf{Exp}^{CMS^{\ell,\ell\text{-}}}}$

$$\begin{array}{lcl}
\mathsf{T}_{\mathcal{L},\ell}(\mathsf{class}\ c\ \{md_1 \ldots md_n\}) & = & \mathsf{class}\ c\ \{\mathsf{T}_{\mathcal{L},\ell}(md_1) \ldots \mathsf{T}_{\mathcal{L},\ell}(md_n)\} \\
\mathsf{T}_{\mathcal{L},\ell}(\mathsf{static}\ t\ m\ (t_1\ x_1, \ldots, t_n\ x_n)\,\{e\}) & = & \mathsf{static}\ \mathsf{T}_{\mathcal{L},\ell}(t)\ m\ (\mathsf{T}_{\mathcal{L},\ell}(t_1)\ x_1, \ldots, \mathsf{T}_{\mathcal{L},\ell}(t_n)\ x_n)\,\{\mathsf{T}_{\mathcal{L},\ell}(e)\}
\end{array}$$

$\boxed{\mathsf{T}_{\mathcal{L},\ell} : \mathsf{Type}^{\mathsf{MCL}} \to \mathsf{Exp}^{CMS^{\ell,\ell\text{-}}}}$

$$\begin{array}{lcll}
\mathsf{T}_{\mathcal{L},\ell}(c) & = & x(\ell_1,c_1); \ldots; x(\ell_n,c_n) & \text{if } \mathsf{dom}(\mathcal{L}(\ell,c)) = \{(\ell_i,c_i) \mid i \in 1..n\} \\
\mathsf{T}_{\mathcal{L},\ell}(\mathsf{bool}) & = & \mathsf{bool} & \\
\mathsf{T}_{\mathcal{L},\ell}(\mathsf{int}) & = & \mathsf{int} &
\end{array}$$

$\boxed{\mathsf{T}_{\mathcal{L},\mathcal{S}} : \mathsf{Exp}^{\mathsf{MCL}} \to \mathsf{Exp}^{CMS^{\ell,\ell\text{-}}}}$

$$\begin{array}{lcl}
\mathsf{T}_{\mathcal{L},\mathcal{S}}(x) & = & x \\
\mathsf{T}_{\mathcal{L},(\mathcal{S},\ell)}(\mathsf{new}\ c()) & = & \mathsf{new}\ \mathsf{T}_{\mathcal{L},\ell}(c)() \\
\mathsf{T}_{\mathcal{L},(\mathcal{S},\ell)}(c.m(e_i^{\,i\in n})) & = & \mathsf{T}_{\mathcal{L},\ell}(c).m(e'^{\,i\in n}_i),\ \text{with } \mathsf{T}_{\mathcal{L},(\mathcal{S},\ell)}(e_i) = e'_i, i \in 1..n \\
\mathsf{T}_{\mathcal{L},(\mathcal{S},\ell)}(\mathsf{ret}\,(e)) & = & \mathsf{T}_{\mathcal{L},\mathcal{S}}(e) \\
\mathsf{T}_{\mathcal{L},\mathcal{S}}(e_1 + e_2) & = & e'_1 + e'_2,\ \text{with } \mathsf{T}_{\mathcal{L},\mathcal{S}}(e_i) = e'_i, i \in \{1,2\} \\
\mathsf{T}_{\mathcal{L},\mathcal{S}}(\mathsf{if}\,(e_1)\,\{e_2\}\,\mathsf{else}\{e_3\}) & = & \mathsf{if}\,(e'_1)\,\{e'_2\}\,\mathsf{else}\{e'_3\},\ \text{with } \mathsf{T}_{\mathcal{L},\mathcal{S}}(e_i) = e'_i, i \in \{1,2,3\} \\
\mathsf{T}_{\mathcal{L},\mathcal{S}}(n) & = & n \\
\mathsf{T}_{\mathcal{L},\mathcal{S}}(b) & = & b,\ \text{with } b = \mathtt{true}, \mathtt{false}
\end{array}$$

Figure 13: Translation

Output assignment The output assignment keeps track of all the existing classes, corresponding to pairs (ℓ_d, c) with ℓ_d defining loader. For each (name corresponding to) a pair (ℓ_d, c) with ℓ_d defining loader, there is an output component mapping the name $X(\ell_d, c)$ into the corresponding (translated) class declaration. Recall that $\lambda^{\mathsf{CDec}}_{p,\mathcal{U}}(\ell, c) = cd$ stands for $(\ell.\mathsf{loadClass}(c), \emptyset) \xrightarrow[p,\mathcal{U}]{\star} ((\_,\_), cd)$.

Local assignment The local part of the configuration keeps track of the loading requests which are already in the cache. For each (variable corresponding to) a loading request (ℓ, c) in L, there is a local component which maps $x(\ell, c)$ into the (translated) class declaration associated in L with this loading request.

Low-priority link All input names are linked (by using the low-priority link operator link^{-}).

The translation of a class declaration simply propagates to method return and argument types and bodies.

A class name c is translated into an expression $x(\ell_1, c_1); \ldots; x(\ell_n, c_n)$ which keeps track of the loading requests (ℓ_1, c_1), $\ldots$, (ℓ_n, c_n) which are resolved as an effect of the loading request (ℓ, c), with ℓ current loader for the code which contains c. Note that the loading requests can be taken in an arbitrary sequence, since all the classes that are loaded as an effect of the resolution of a given request are added to the current cache in one (user-level) reduction step. In this way, all variables corresponding to these classes have to be linked, correctly simulating what happens in MCL.

Translation of expressions is parametrized by the current stack of loaders, which is used for taking the right initiating loader in translating class names inside expressions. Indeed, recall that loaders are pushed onto the stack each time the evaluation of a method body starts and that, during this evaluation, method body is boxed into a ret operator. Hence, the initiating loader must be determined starting from the bottom of the stack and going up to the next (upper) one in $\mathcal{S}$ each time a $\mathsf{ret}\,(e)$ expression is encountered (this is obtained, in the translation of $\mathsf{ret}\,(e)$, by depriving the stack $\mathcal{S}$ of its bottom element in the recursive call on e).

The given translation preserves the semantics, formally:

Theorem 5.1 *If* $p,\mathcal{U} \vdash (\mathcal{S};\ L;\ e)$ *and* $(\mathcal{S};\ L;\ e) \xrightarrow[p,\mathcal{U}]{} (\mathcal{S}';\ L';\ e')$, *then* $\mathsf{T}_{p,\mathcal{U}}(\mathcal{S};\ L;\ e) \rightarrow^* \mathsf{T}_{p,\mathcal{U}}(\mathcal{S}';\ L';\ e')$.

Proof *By induction on the derivation of the reduction and case-analysis on the last rule applied in the derivation of the typing judgement.* ∎

Although we cannot state and prove it formally because for lack of space we have not presented the type system for CMS^{ℓ,ℓ^-}, the translation preserves also well-formedness of terms (that is, transforms well-formed MCL user states into well-formed CMS^{ℓ,ℓ^-} configurations).

6 CONCLUSION

We have presented a toy language, called MCL, which embodies the following features of Java: class loading is dynamically triggered when the first reference to a class name is encountered, and the actual class which is loaded is not uniquely determined by the class name, since different class loaders can be used in the same application.

Formal descriptions of Java class loading are given already in, e.g., the already cited [13], and in [15, 7]. With respect to these models, the aim of this paper is different for at least two reasons.

First, in the same spirit as in previous papers on selected Java features [9, 4], we wanted to model just the two features mentioned above in isolation, abstracting from the complexity of the language and other orthogonal aspects such as bytecode verification[2] and reflection. Indeed, we believe these two features constitute, in a sense, the essence of the way dynamic linking of fragments is allowed in Java.

Second, our aim here is also on the design side; indeed, we have shown in MCL that, besides a rigid approach where all classes are loaded by a unique loader whose behavior just depends on a user defined class path, and one where arbitrary loaders can be created and manipulated by the user, an intermediate solution is also possible based on stratification, in the sense that the configuration language can influence the execution of the user program, but not conversely. We believe that this possibility is interesting since it allows to statically check safety while retaining some flexibility, and should be further investigated in the context of real programming languages.

Then, we have defined an encoding of MCL into a kernel calculus CMS^{ℓ,ℓ^-} [2] which models various forms of linking, and proved that the translation is correct in the sense that it preserves the language semantics. In this way we have provided an extended application showing the effectiveness of CMS^{ℓ,ℓ^-} for modeling linking policies of real languages.

The aim of the translation given in this paper is not to show the full expressiveness of CMS^{ℓ,ℓ^-}, since only some features of the calculus are actually needed for the encoding, but rather, to analyze Java-like linking within a general framework with different linking operators, and to serve as a basis for studying and comparing possible variations of the Java-like mechanism.

Acknowledgments. This work has been partially supported by Dynamic Assembly, Reconfiguration and Type-checking - EC project IST-2001-33477, and APPSEM II - Thematic network IST-2001-3895. We warmly thank Davide Ancona for the previous common work in [2] and other discussions which directly inspired this paper, Giovanni Lagorio for many precious discussions and suggestions, and Alex Buckley and an anonymous referee for their careful reading and useful comments.

[2] Which, however, affects the loading mechanism in what some classes need to be loaded earlier.

REFERENCES

[1] D. Ancona, S. Fagorzi, and E. Zucca. A calculus for dynamic linking. In C. Blundo and C. Laneve, editors, *Italian Conf. on Theoretical Computer Science 2003*, number 2841 in Lecture Notes in Computer Science, pages 284–301, 2003.

[2] D. Ancona, S. Fagorzi, and E. Zucca. A calculus for dynamic reconfiguration with low priority linking. In *WOOD'04: Workshop on Object-Oriented Developments*, August 2004.

[3] D. Ancona, S. Fagorzi, and E. Zucca. A calculus with lazy module operators. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *TCS 2004 (IFIP Int. Conf. on Theoretical Computer Science)*, pages 423–436. Kluwer Academic Publishers, 2004.

[4] D. Ancona, G. Lagorio, and E. Zucca. A core calculus for Java exceptions. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*, SIGPLAN Notices. ACM Press, October 2001.

[5] D. Ancona and E. Zucca. A calculus of module systems. *Journ. of Functional Programming*, 12(2):91–132, 2002.

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

[7] S. Drossopoulou. Towards an abstract model of Java dynamic linking and verfication. In R. Harper, editor, *TIC'00 - Third Workshop on Types in Compilation (Selected Papers)*, volume 2071 of *Lecture Notes in Computer Science*, pages 53–84. Springer, 2001.

[8] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 41–82. Springer, 1999.

[9] A. Igarashi, B. 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] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1998*, volume 33(10) of *SIGPLAN Notices*, pages 36–44. ACM Press, October 1998.

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

[12] E. Machkasova and F.A. Turbak. A calculus for link-time compilation. In *ESOP 2000 - European Symposium on Programming 2000*, number 1782 in Lecture Notes in Computer Science, pages 260–274. Springer, 2000.

[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, October 2000.

[14] V. Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997. http://www.research.att.com/~vj/bug.html.

[15] A. Tozawa and M. Hagiya. Formalization and analysis of class loading in Java. *Higher-Order and Symbolic Computation*, 15:7–55, March 2002.

[16] J.B. Wells and R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules. In *ESOP 2000 - European Symposium on Programming 2000*, number 1782 in Lecture Notes in Computer Science, pages 412–428. Springer, 2000.

About the authors

Sonia Fagorzi Ph.D. student in Computer Science at the University of Genova since February 2002. Her research interests are in the area of programming languages; in particular, design and foundations of modular and object-oriented languages and systems. She can be reached at fagorzi@disi.unige.it.
See also http://www.disi.unige.it/person/FagorziS/.

Elena Zucca Associate professor at the University of Genova since 1999, previously assistant professor at the University of Genova since 1989. Author of more than 40 papers in international journals and conferences. Her main research contributions are in the semantics and specification of concurrent and object-oriented languages, extension of algebraic techniques to dynamic systems, module calculi, type systems and semantics of Java-like languages. She can be reached at zucca@disi.unige.it.
See also http://www.disi.unige.it/person/ZuccaE/.