Modeling Multiple Class Loaders by a Calculus for Dynamic Linking*

Sonia Fagorzi
DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy
fagorzi@disi.unige.it

Elena Zucca
DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy
zucca@disi.unige.it

Davide Ancona
DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy
davide@disi.unige.it

ABSTRACT

In a recent paper we proposed a calculus for modeling dynamic linking independently of the details of a particular programming environment.

Here we use a particular instantiation of this calculus to encode a toy language, called MCL, which provides an abstract view of the mechanism of dynamic class loading with multiple loaders as in Java.

The aim is twofold. On one hand, we show an example of application of the calculus in modeling existing loading and linking policies, showing in particular that Java-like loading with multiple loaders can be encoded without exploiting the full expressive power of the calculus. On the other hand, we provide a simple formal model which allows a better understanding of Java-like loading mechanisms and also shows an intermediate solution between the rigid approach based only on the class path and that which allows arbitrary user-defined loaders, which can be intricate and error-prone.

Keywords

Dynamic linking, multiple loaders, Java

1. INTRODUCTION

In a recent paper [1] we proposed a calculus (called CDL for Calculus for Dynamic Linking in the sequel) for modeling dynamic linking independently of the details of a particular programming environment.

The calculus distinguishes at the language level the two phases of software configuration and execution, by introducing separate syntactic notions of linkset expression and command, respectively. More precisely, terms of CDL are *configurations*, which are pairs consisting of a linkset expression and a command. Configurations can evolve in two ways: either by simplifying the linkset expression (that is, performing a configuration step) or by performing a step in the execution of the command. However, configuration and execution phases are interleaved: an execution step may trigger a configuration step (for instance, when a not yet linked fragment is needed) and modify the execution context (for instance, updating a fragment).

Here we use a particular instantiation of this calculus to encode a toy language, called MCL, which provides an abstract view of the mechanism of dynamic class loading with multiple loaders as in Java [8, 7, 9].

The aim is twofold. On one hand, we show an example of application of the calculus for modeling existing loading and linking policies, showing in particular that Java-like loading with multiple loaders can be encoded without exploiting the full expressive power of the calculus (see the Conclusion). On the other hand, we provide a simple formal model which allows to understand Java-like loading mechanisms.

Terms in MCL model intermediate steps in the execution of an application. Execution consists in the evaluation of an expression in a very 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 needed (in our simple language, an instance creation or a 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. In addition to considering a very simple language, we also do not model bytecode verification, and the Java approach based on reflection which allows the programmer to use arbitrary class loaders by defining subclasses of the class `ClassLoader`.

One motivation is that, as done in previous papers for other Java features (e.g., inheritance and late binding in [6], checked exceptions in [2]), 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 a configuration language consisting in a fixed set of loaders, formally modeled by what we call a *loading environment*. This approach is less flexible w.r.t. the Java behavior, where both aspects are part of the language, hence loaders might change during execution; here, instead, program execution cannot affect the loading environment. On the other side,

*Partially supported by Dynamic Assembly, Reconfiguration and Type-checking - EC project IST-2001-33477, APPSEM II - Thematic network IST-2001-38957, and Murst NAPOLI - Network Aware Programming: Oggetti, Linguaggi, Implementazioni.

SAC'04 March 14-17, 2004, Nicosia, Cyprus

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 rest of the paper is organized as follows. In Section 2 we formally define MCL (in 2.1 the syntax, in 2.2 the reduction rules modeling execution). In Section 3 we briefly recall the calculus CDL introduced in [1]. In Section 4 we define a translation from MCL into CDL 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 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 $\mathsf{dom}(f)$; $a_1 : b_1, \ldots, a_n : b_n$ denotes the function which returns b_i on a_i, is undefined otherwise. We write finite sequences in the form either $a_1, \ldots, a_n$ or $a_i^{1 \in 1..n}$.

2.1 Syntax

The syntax of the language is given in Fig.1.

Metavariables $\ell \in \mathsf{Loader}$, $c \in \mathsf{CName}$, m and x range over primitive sets of *loaders*, *class names*, *method names*, and *variables*, respectively.

A loading environment λ is a function that, given a loader ℓ and a class name c (that is, a symbolic reference to a class which appears in code), returns a pair consisting of another loader ℓ_d and a class definition cd. We denote ℓ_d and cd by $\lambda_{\mathsf{Loader}}(\ell, c)$ and $\lambda_{\mathsf{CDef}}(\ell, c)$, respectively. The mapping from ℓ into ℓ_d abstractly models the delegation relationship between the *initiating* and the *defining* loader for a class. Indeed, in Java, when a class name which has not been resolved yet is encountered, the current loader (called the initiating loader), that is, that which has been used to load the code currently in execution, can delegate the task to another loader, until a loader (called the defining loader) actually loads the class. The mapping from ℓ into cd models the actual loading step (method `defineClass` in user-defined loaders, see [7]).

Of course, if the defining loader ℓ_d obtained for a certain class name c is taken as initiating loader for c, then we expect to obtain the same defining loader and class definition. Formally we have the following requirement on λ:

$$\text{if } \lambda_{\mathsf{Loader}}(\ell, c) = \ell_d, \text{ then } \lambda(\ell_d, c) = \lambda(\ell, c).$$

Snapshots of the execution of an application are modeled by *configurations*, which consist of three components: a stack of loaders, where the top of the stack corresponds to the loader which has been used to load the code currently in execution, a *loaded class cache* which is a mapping of the same kind of the loading environment which records the encountered loading requests [8, 7, 9], and the expression currently in execution.

A class definition 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. Method headers contain types for keeping a Java-like syntax (we do not consider primitive types for simplicity, so types are just class names); however, in this paper we do not deal with type-checking.

Expressions which appear in user-defined code are of four kinds: variable, instance creation,(static) method invocation, and let-in. As usual, in reduction semantics we will also use expressions modeling intermediate steps of the evaluation, in this case expressions of the form $\mathsf{ret}\,(e)$ representing the evaluation of a method body inside an invoking expression (see rules (M2) and (R2) in Fig.2).

Values (that is, expressions which cannot be reduced, see next section) are instance creation expressions.

2.2 Semantics

The semantics of MCL is modeled by a rewriting relation on configurations, parameterized by a loading environment assumed to be be fixed during the execution, formally:

$$\xrightarrow[\lambda]{} \subseteq \mathsf{Conf} \times \mathsf{Conf}.$$

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 configuration* where these preliminary steps have already been performed, hence the stack of loaders just contains ℓ_d, and the domain of the loaded class cache contains two elements (possibly coinciding), (ℓ, c) and (ℓ_d, c).

Formally, initial configurations will have shape $(\ell_d;\ L;\ e)$, with

- either L of the form $(\ell_d, c) : cd$, with $\lambda(\ell_d, c) = (\ell_d, cd)$
- or L of the form $(\ell_d, c) : cd, (\ell, c) : cd$, with $\lambda(\ell, c) = \lambda(\ell_d, c) = (\ell_d, cd)$ and $\ell \neq \ell_d$.

Reduction rules are shown in Fig.2.

Rule (C) models a class loading step, which is performed when a class name c is encountered with current loader ℓ and the loading request (ℓ, c) has not been considered yet (the only two cases are instance creation and method invocation). In this case, the loaded class cache L is updated, by adding the association from (ℓ, c) into the corresponding defining loader ℓ_d and class definition cd in λ, and, if (ℓ_d, c) is not already present (this means that it was not present in L and that ℓ_d is different from ℓ), the same association from (ℓ_d, c).

Rules (M1) and (M2) deal with invocation of a method of an already loaded class. Rule (M1) models evaluation of arguments from left to right. Rule (M2) models the method invocation step: first, the loaded class cache is used to get the defining loader ℓ_d and the class definition associated to the current loader ℓ and the class name c; then, the current expression to be evaluated is updated to the body of the invoked method where formal parameters have been replaced by arguments, and the ret operator has been applied (see rule (R2)). The current loader is updated to the defining loader of the class to which the invoked method belongs, that is, ℓ_d is put on top of the stack of loaders.

Rules (R1) and (R2) deal with method evaluation. Rule (R1) is just a propagation rule. Rule (R2) 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.

$\lambda : \mathsf{Loader} \times \mathsf{CName} \xrightarrow{fin} \mathsf{Loader} \times \mathsf{CDef}$			**loading environments**
$\gamma \in \mathsf{Conf}$	::=	$(\mathcal{S};\ L;\ e)$ with	**configurations**
		$\mathcal{S} \triangleq \ell_i{}^{i \in 1..n}$	loaders stack
		$L : \mathsf{Loader} \times \mathsf{CName} \xrightarrow{fin} \mathsf{Loader} \times \mathsf{CDef}$	loaded class cache
$cd \in \mathsf{CDef}$	::=		**class definitions**
		$\mathsf{class}\ c\ \{$ $\mathsf{static}\ c_j\ m_j\ \left(c_j^1\ x_j^1, \ldots, c_j^{n_j}\ x_j^{n_j}\right) \{e_j\}^{j \in 1..p}$ $\}$	
$e \in \mathsf{Exp}$	::=		**expressions**
		x	variable
	|	$\mathsf{new}\ c()$	instance creation
	|	$c.m(e_i{}^{i \in n})$	method invocation
	|	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	let-in
	|	$\mathsf{ret}\,(e)$	method evaluation
$v \in \mathsf{Val}$	::=	$\mathsf{new}\ c()$	**values**

Figure 1: MCL syntax

Rules (L1) and (L2) are the standard rules for the let-in construct with a call-by-value semantics.

3. A CALCULUS OF LINKSETS

In this section we briefly present the calculus CDL which we will use for encoding MCL. The reader can refer to [1] for all technical details and an extended discussion on motivations.

Terms of the calculus are called *configurations*, model snapshots in the lifetime of a software system, and are pairs consisting of a *linkset* and a *command*, corresponding to two different phases called *configuration phase* and *execution phase*, respectively. The configuration phase corresponds to the process of obtaining an executable application by combining in various ways different pieces of software. This phase may be interleaved with the execution phase, even though it typically takes place before execution; for instance, it can correspond to what is performed by a (static) linker. In CDL, this phase is modeled by the fact that the linkset is, in general, a complex expression, which must be reduced to a normal form for performing some kind of command; after that, it is possible to start reducing the command, in the context provided by the linkset.

Basic linksets are collections of named, interdependent code fragments, and operators for composing linksets correspond to operations one can perform during configuration. In particular, an important operation one can perform on a linkset is to *link* a fragment, say X, that is, to resolve the dependencies on X. *Static linking* (as modeled, e.g., by Cardelli [4]) requires fully linked linksets in order to start the execution of any command, so that all dependencies have been resolved before the execution phase starts. In CDL, instead, linking can take place at run-time too. Hence, execution can start also when the linkset is not fully linked; during execution of the command, we can find references to other fragments which have not been resolved yet, hence they need to be dynamically linked. Two forms of run-time linking are considered: in the first form (*permanent* dynamic linking), a fragment is permanently linked to the executing program the first time it is needed. In the second form (*volatile* dynamic linking), a fragment is made available to the executing program when its code is needed, but not permanently linked, so that when a later reference to the same fragment is encountered the linking must be performed again, and in case the code of the fragment has been changed thereafter the new version is used.

An important feature of CDL is that it is *parametric* in the particular language used for writing code in single fragments (the *core* language following the terminology used in module systems). In the next section, in order to encode MCL, we will define a particular instantiation of the calculus.

3.1 Syntax

The syntax of the calculus is given in Fig.3.

The metavariables $X \in \mathsf{Name}$ and $x \in \mathsf{Var}$ range over (fragment) *names* X and *variables* x, respectively. Elements of Exp are (core) expressions, that is, the expressions of the underlying language used for defining single code fragments. Intuitively, names are used to refer to fragments from outside a linkset, whereas variables are used in code within a linkset (indeed, expressions are assumed to be built on top of variables, see the production for Exp).

A basic linkset consists of three components. The ι component is a mapping from variables into names and corresponds to the *input* linkset fragments; the o component is a mapping from names into expressions and corresponds to the *output* fragments; finally, the ρ component is a mapping from variables into expressions and represents the *local* (that is, already linked) fragments. Variables in the domain of ι and ρ are called the *deferred* and the *local* variables of the basic linkset, respectively. The sets of deferred and local variables must be disjoint.

There are four operators on linksets: the *sum* operator, which allows merging of two linksets, and three different *link* operators: link for static linking, $\mathsf{dlink}^{\mathsf{p}}$ for permanent dynamic linking and $\mathsf{dlink}^{\mathsf{v}}$ for volatile dynamic linking. We will explain linkset operators in more detail when introducing reduction rules for linksets and configurations.

The linkset expression $\mathsf{dlink}_{X_1}^{\mathsf{K}_1}(\ldots(\mathsf{dlink}_{X_n}^{\mathsf{K}_n}(l))\ldots)$, where $n \geq 0$, $K_i \in \{\mathsf{p}, \mathsf{v}\}$, is abbreviated by $\mathsf{dlink}_{[P;V]}(l)$, where $P = \{X_i \mid \mathsf{K}_i = \mathsf{p}\}$ and $V = \{X_1, \ldots, X_n\} \backslash P$ (so, in the case $n = 0$, $\mathsf{dlink}_{[\emptyset;\emptyset]}(l)$ obviously coincides with l). This is sound

(C) $\dfrac{}{(\ell, \mathcal{S};\ L;\ e) \xrightarrow[\lambda]{} (\ell, \mathcal{S};\ L';\ e)}$ $\begin{array}{l} e ::= \mathsf{new}\ c() \mid c.m({v_i}^{i\in 1..n}) \\ (\ell, c) \notin \mathsf{dom}(L) \end{array}$

where, setting $\ell_d = \lambda_{\mathsf{Loader}}(\ell, c)$, L' is defined by:

$$L' = \begin{cases} L, (\ell, c) : \lambda(\ell, c), (\ell_d, c) : \lambda(\ell, c) & \text{if } \ell \neq \ell_d \text{ and } (\ell_d, c) \notin \mathsf{dom}(L') \\ L, (\ell, c) : \lambda(\ell, c) & \text{otherwise} \end{cases}$$

(M1) $$\frac{(\mathcal{S};\ L;\ e_i) \xrightarrow[\lambda]{} (\mathcal{S}';\ L';\ e_i')}{(\mathcal{S};\ L;\ c.m(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n)) \xrightarrow[\lambda]{} (\mathcal{S}';\ L';\ c.m(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n))}$$

(M2) $$\frac{}{\left(\ell, \mathcal{S};\ L;\ c.m({v_i}^{i\in 1..n})\right) \xrightarrow[\lambda]{} \left(\ell_d, \ell, \mathcal{S};\ L;\ \mathsf{ret}\left(e_k\{x_k^i : {v_i}^{i\in 1..n}\}\right)\right)} \ (\ell, c) \in \mathsf{dom}(L)$$

where:

- $L_{\mathsf{Loader}}(\ell, c) = \ell_d$
- $L_{\mathsf{CDef}}(\ell, c) =$ $\mathsf{class}\ c\ \{$ $\mathsf{static}\ c_j\ m_j\ \left(c_j^1\ x_j^1, \ldots, c_j^{n_j}\ x_j^{n_j}\right)\ \{e_j\}^{j\in 1..p}$ $\}$
- $m = m_k$ and $n = n_k$, $k \in 1..p$

(R1) $$\frac{(\mathcal{S};\ L;\ e) \xrightarrow[\lambda]{} (\mathcal{S}';\ L';\ e')}{(\mathcal{S};\ L;\ \mathsf{ret}\,(e)) \xrightarrow[\lambda]{} (\mathcal{S}';\ L';\ \mathsf{ret}\,(e'))}$$

(R2) $$\frac{}{(\ell, \mathcal{S};\ L;\ \mathsf{ret}\,(v)) \xrightarrow[\lambda]{} (\mathcal{S};\ L;\ v)}$$

(L1) $$\frac{(\mathcal{S};\ L;\ e_1) \xrightarrow[\lambda]{} (\mathcal{S}';\ L';\ e_1')}{(\mathcal{S};\ L;\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \xrightarrow[\lambda]{} (\mathcal{S}';\ L';\ \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2)}$$

(L2) $$\frac{}{(\mathcal{S};\ L;\ \mathsf{let}\ x = v\ \mathsf{in}\ e_2) \xrightarrow[\lambda]{} (\mathcal{S};\ L;\ e_2\{x : v\})}$$

Figure 2: MCL reduction rules

since semantics of a linkset expression is invariant w.r.t. to permutations and repetitions in a sequence of application of the dynamic link operators, and, moreover, in case of application of both a permanent and a volatile dynamic link operator for the same name, only the permanent one is taken into account (see the reduction rules in Fig.5).

Intuitively, commands model actions which can be performed in the execution phase, which include standard execution of the underlying core expressions (hence expressions are included into commands) and metaoperations on fragments which can be interleaved with standard execution. Examples of metaoperations are a set operation which updates the code of an existing fragment, a get operation which loads the code of some fragment, and the operation stop for execution termination.

Expressions of the core language are not specified; we only assume that they contain variables. In the next section we will provide a definition of Exp for the particular instantiation of CDL we use for translating MCL.

A configuration is a pair consisting of a linkset and a command. Note that the command in a configuration may contain both variables (since expressions may contain variables) and names of the current linkset. Indeed, on one hand an execution step can be a standard execution step, that is, an evaluation step of an expression. Code in execution can refer to *internal* names of fragments, either already resolved (local variables in ρ), or still to be resolved (deferred variables in ι). In modeling concrete languages, as in the translation of next section, variables will correspond to identifiers appearing in code, e.g., class names (for a given current loader), and a class name not loaded yet will be a deferred variable. On the other hand, an execution step can be a metaoperation which manipulates the fragment "from the outside", hence through names. Names will correspond to physical names (in the file system or on the web) and ι to the mechanism used by a loader for associating to a symbolic name a physical name. A set command, for instance, models the fact that the code contained in some file is modified, either by effect of an external agent or even by the application itself.

3.2 Semantics

We define two different reduction relations $\xrightarrow[\mathsf{conf}]{}$ and $\xrightarrow[\mathsf{exec}]{}$, corresponding to the configuration and execution

$l \in \mathsf{Linkset}$	::=		**linksets**
		$[\iota; o; \rho]$ with $\mathsf{dom}(\iota) \cap \mathsf{dom}(\rho) = \emptyset$	basic linkset
	|	$l_1 + l_2$	sum
	|	$\mathsf{link}_X\ (l)$	link
	|	$\mathsf{dlink}^{\mathsf{p}}_X\ (l)$	(permanent) dynamic link
	|	$\mathsf{dlink}^{\mathsf{v}}_X\ (l)$	volatile (dynamic) link
$\iota : \mathsf{Var} \xrightarrow{fin} \mathsf{Name}$			**input** assignment
$o : \mathsf{Name} \xrightarrow{fin} \mathsf{Exp}$			**output** assignment
$\rho : \mathsf{Var} \xrightarrow{fin} \mathsf{Exp}$			**local** assignment
$c \in \mathsf{Com}$	::=	$e \mid \mathsf{set}(X, e) \mid \mathsf{get}(X) \mid \mathsf{stop} \mid \ldots$	**commands**
$e \in \mathsf{Exp}$	::=	$x \mid \ldots$	**(core) expressions**
$\gamma \in \mathsf{Conf}$	::=	$(l,\ c)$	**configurations**

Figure 3: CDL syntax

phase, respectively. By definition, these two relations are defined over well-formed terms, so we have omitted all side conditions ensuring well-formedness of terms.

The reduction rules for the configuration phase are given in Fig.4 (we omit rules for contextual closure which can be found in [1]). In this phase only the linkset expression in a configuration is reduced.

The reduction rule for the sum is similar to those of module and link calculi (see, e.g., [3]). This operation has the effect of gluing together two linksets. The deferred and local variables of one linkset must be disjoint from those of the other and, moreover, the sets of output fragments of the two linksets must be disjoint ($\mathsf{dom}(o_1) \cap \mathsf{dom}(o_2) = \emptyset$). Both these conditions are implicit, since reduction is defined only over well-formed terms; the first condition can always be satisfied by an appropriate α-conversion, while in the second case the conflict cannot be resolved by an α-conversion and the reduction gets stuck. The sets of input fragments of the two linksets can have a non empty intersection and the resulting set of input fragments of the sum is simply the union of them; this means that imported fragments with the same name in the two linksets are shared.

Finally, in the sum the sets of fragment names dynamically linked is obtained by taking the union of the corresponding sets in the two linksets, and taking only the permanent operator into account when both permanent and volatile linking turn out to be applied to the same name.

In rule (link), the effect of linking fragment X is that this fragment name is resolved, hence it disappears from the input names and all the variables mapped by ι into it are now linked, that is, they become local. Here $\iota \setminus L$ denotes the restriction of the partial function ι to the variables which are not in L. These variables are associated with the definition of X in the output assignment, which must exist (side-condition). Moreover, the name X also disappears from those for which a dynamic linking operator is applied.

Note that there are no reduction rules for the dynamic link operators; indeed, the intuition for these operators is that they are not performed during the configuration phase, but they will be performed on demand only after execution is started (see reduction rules for configurations in Fig.5). As a consequence, normal forms w.r.t. the configuration relation are obtained by a sequence of dynamic linking operators applied to a basic linkset, that is, are of the form $\mathsf{dlink}_{[P;V]}\ ([\iota; o; \rho])$.

The reduction rules for the execution phase are given in Fig.5 (we omit rules for contextual closure which can be found in [1]).

The reduction relation $\xrightarrow[\mathsf{exec}]{}$ is parametric on the relation $\xrightarrow[\mathsf{core}]{}$ which corresponds to evaluation of core expressions.

The (linkset) and (core) rules express that an execution step can consist in a configuration step of the linkset or, if the command is a core expression, in an evaluation step at the core level.

The subsequent three rules can only be applied when the linkset is in normal form.

The (set) rule expresses that an execution step can consist in updating the definition of an existing output fragment; note that this execution step modifies both the linkset and the command in the current configuration.

The (get) rule expresses that an execution step can consist in obtaining as command to be executed the current definition of an existing fragment.

The (var) rule shows how dynamic and volatile linking work. If a variable x is defined in ρ, then its corresponding code is already available and does not need to be linked; on the other hand, if x is deferred (that is, the corresponding fragment, say X, has not been linked yet), then, in the command, the variable is replaced by the current definition of X. Moreover, if the linking is dynamic, then the corresponding fragment X is permanently linked in the linkset, so that further occurrences of x will always refer to the same definition, while this is not the case if the linking is volatile.

4. TRANSLATION

As target language we consider a particular instantiation of CDL (see Fig.6).

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), instance creations, method invocations, let-in expressions and class defi-

$$\text{(sum)}\ \frac{}{\mathsf{dlink}_{[P_1;V_1]}\,(l_1) + \mathsf{dlink}_{[P_2;V_2]}\,(l_2) \xrightarrow[\mathsf{conf}]{} \mathsf{dlink}_{[P;V\setminus P]}\,([\iota_1,\iota_2;o_1,o_2;\rho_1,\rho_2])} \quad \begin{array}{l} V = V_1 \cup V_2 \\ P = P_1 \cup P_2 \\ l_i \equiv [\iota_i;o_i;\rho_i], i \in \{1,2\} \end{array}$$

$$\text{(link)}\ \frac{}{\mathsf{link}_X\left(\mathsf{dlink}_{[P;V]}\,([\iota;o;\rho])\right) \xrightarrow[\mathsf{conf}]{} \mathsf{dlink}_{[P\setminus\{X\};V\setminus\{X\}]}\left([\iota\setminus L;o;\rho,x:o(X)^{x\in L}]\right)} \quad \begin{array}{l} L = \{x \mid \iota(x) = X\} \\ L \neq \emptyset \implies X \in \mathsf{dom}(o) \end{array}$$

Figure 4: CDL reduction rules for the configuration phase

$$\text{(linkset)}\ \frac{l \xrightarrow[\mathsf{conf}]{} l'}{(l,\ c) \xrightarrow[\mathsf{exec}]{} (l',\ c)} \qquad \text{(core)}\ \frac{e \xrightarrow[\mathsf{core}]{} e'}{(l,\ e) \xrightarrow[\mathsf{exec}]{} (l,\ e')}$$

$$\text{(set)}\ \left(\mathsf{dlink}_{[P;V]}\,([\iota;o;\rho])\,,\ \mathsf{set}(X,e)\right) \xrightarrow[\mathsf{exec}]{} \left(\mathsf{dlink}_{[P;V]}\,([\iota;o\{X:e\};\rho])\,,\ \mathsf{stop}\right),\quad X \in \mathsf{dom}(o)$$

$$\text{(get)}\ \left(\mathsf{dlink}_{[P;V]}\,([\iota;o;\rho])\,,\ \mathsf{get}(X)\right) \xrightarrow[\mathsf{exec}]{} \left(\mathsf{dlink}_{[P;V]}\,([\iota;o;\rho])\,,\ o(X)\right),\quad X \in \mathsf{dom}(o)$$

$$\text{(var)}\ \left(\mathsf{dlink}_{[P;V]}\,([\iota;o;\rho])\,,\ x\right) \xrightarrow[\mathsf{exec}]{} \begin{cases} \left(\mathsf{dlink}_{[P;V]}\,([\iota;o;\rho])\,,\ \rho(x)\right) & \text{if } x \in \mathsf{dom}(\rho) \\ \left(\mathsf{link}_X\left(\mathsf{dlink}_{[P;V]}\,([\iota;o;\rho])\right)\,,\ o(X)\right) & \text{if } \iota(x) = X \wedge X \in (\mathsf{dom}(o) \cap P) \\ \left(\mathsf{dlink}_{[P;V]}\,([\iota;o;\rho])\,,\ o(X)\right) & \text{if } \iota(x) = X \wedge X \in (\mathsf{dom}(o) \cap V) \end{cases}$$

Figure 5: CDL reduction rules for the execution phase

nitions. Class definitions 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 definition, by applying rule (var) of CDL. Note that class definitions contain the name of the class being declared for keeping a Java-like syntax; however, this information is useless since we do not deal with verification. Core rules include method invocation and let-in (values v are new applications to class definitions).

The translation is defined in Fig.7. We use the superscripts MCL and CDL to denote syntactic categories of MCL and CDL, respectively, when there is ambiguity.

A MCL configuration is translated into a CDL configuration (linkset expression and command); the translation is parameterized on a fixed loading environment.

The linkset expression consists of a linkset in normal form $\mathsf{dlink}^{\mathsf{p}}_P\,([\iota;o;\rho])$ defined as follows.

Input fragments The input part of the linkset keeps trace 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.

Output fragments The output part of the linkset keeps trace of the existing classes, corresponding to pairs (ℓ_d, c) with ℓ_d defining loader. For each (name corresponding to) a loading request (ℓ_d, c) with ℓ_d defining loader, there is an output component mapping the name $X(\ell_d, c)$ into a class definition. This class definition is obtained from that in λ, by replacing each class name[1] c' by (the variable corresponding to) c' and its initiating loader, which is the defining loader for the loading request (ℓ_d, c).

Local fragments The local part of the linkset keeps trace 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 class definition associated in L with this loading request, where the same substitution described above for output components is applied.

Dynamically linked names All output names are dynamically linked (by using $\mathsf{dlink}^{\mathsf{p}}$).

The command to be executed is obtained from the expression e in the MCL configuration, by replacing each class name by the variable corresponding to the class name and its defining loader. The defining loader is obtained from $\lambda_{\mathsf{Loader}}$ using, as usual, the class name and the appropriate initiating loader. In order to determine the initiating loader, 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 case $\mathsf{ret}\,(e)$ of the Com definition, depriving the stack $\mathcal{S}$ of its bottom element in the recursive call on e).

The following theorem states that the translation preserves the semantics.

THEOREM 4.1. *If* $(\mathcal{S};\ L;\ e) \xrightarrow[\lambda]{} (\mathcal{S}';\ L';\ e')$, *then* $\mathsf{T}_\lambda(\mathcal{S};\ L;\ e) \xrightarrow[\mathsf{exec}]{*} \mathsf{T}_\lambda(\mathcal{S}';\ L';\ e')$.

PROOF. *By induction on the derivation of the premise (case analysis on the last* MCL *reduction rule applied).* □

[1]The function CName returns all class names appearing in a class definition.

- $\mathsf{Var} = \{x(\ell, c) \mid \ell \in \mathsf{Loader}, c \in \mathsf{Class}\}$, $\mathsf{Name} = \{X(\ell, c) \mid \ell \in \mathsf{Loader}, c \in \mathsf{Class}\}$
- $e \in \mathsf{Exp} \quad ::= \quad x(\ell, c) \mid \mathsf{new}\ e() \mid e.m(e_1, \ldots e_n) \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid$
  $\mathsf{class}\ c\ \{\mathsf{static}\ c_j\ m_j\ \left(c_j^1\ x_j^1, \ldots, c_j^{n_j}\ x_j^{n_j}\right)\{e_j\}^{j\in 1..p}\}$
  $v \in \mathsf{Val} \quad ::= \quad \mathsf{new\ class}\ c\ \{\mathsf{static}\ c_j\ m_j\ \left(c_j^1\ x_j^1, \ldots, c_j^{n_j}\ x_j^{n_j}\right)\{e_j\}^{j\in 1..p}\}()$
- $$\frac{}{\mathsf{class}\ c\ \{\mathsf{static}\ c_j\ m_j\ \left(c_j^1\ x_j^1, \ldots, c_j^{n_j}\ x_j^{n_j}\right)\{e_j\}^{j\in 1..p}\}.m(v_i{}^{i\in 1..n}) \xrightarrow[\mathsf{core}]{} e_k\{x_k^1 : v_1, \ldots x_k^{n_k} : v_n\}} \quad \begin{array}{l} m = m_k, k \in 1..p \\ n_k = n \end{array}$$

$$\frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e \xrightarrow[\mathsf{core}]{} e\{x : v\}}$$

Figure 6: CDL instantiation

5. CONCLUSION

We have presented a toy language, called MCL, which embodies the following features of Java: class loading is dynamically triggered whenever the first reference to a class name is needed, 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 [9], and in [10, 5]. 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 [6, 2], 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 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 partly 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 defined 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 CDL [1] 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 a first application showing the effectiveness of CDL for modeling linking policies of real languages.

Of course the aim of the translation given in this paper is not to show the full expressiveness of CDL, 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. In particular, this shows the following two facts:

- As expected, Java-like linking can be encoded by a *permanent dynamic link* operator, where a fragment is dynamically but permanently linked to the executing program the first time is needed.
- More important, the current encoding serves as a basis for studying and comparing possible variations of the Java-like mechanism.

An extended version of this paper is in preparation, including also a type system for MCL and a result of preservation of the static semantics.

6. 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, 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.

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

[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. 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.

[6] 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.

[7] 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.

$\mathsf{T}_\lambda : \mathsf{Conf}^{\mathsf{MCL}} \to \mathsf{Conf}^{\mathrm{CDL}}$
$\mathsf{T}_\lambda(\mathcal{S};\ L;\ e) = (\mathsf{dlink}^{\mathsf{p}}_P\ ([\iota; o; \rho])\ ,\ \mathsf{T}_{\lambda,\mathcal{S}}\ (e))$, with:

- $\iota = \{x(\ell, c) : X(\lambda_{\mathsf{Loader}}(\ell, c), c) \mid (\ell, c) \in \mathsf{dom}(\lambda) \setminus \mathsf{dom}(L)\}$
- $o = \{X(\ell_d, c) : \mathsf{T}_{\lambda,\ell_d}\ (cd)\ \mid \lambda(\ell_d, c) = (\ell_d, cd)\}$
- $\rho = \{x(\ell, c) : \mathsf{T}_{\lambda,\ell_d}\ (cd)\ \mid L(\ell, c) = (\ell_d, cd)\}$
- $P \triangleq \mathsf{dom}(o)$

$\mathsf{T}_{\lambda,\ell} : \mathsf{CDef} \to \mathsf{Exp}^{\mathrm{CDL}}$
$\mathsf{T}_{\lambda,\ell}\ (cd) \triangleq cd\{c : x(\lambda_{\mathsf{Loader}}(\ell, c), c) \mid c \in \mathsf{CName}(cd)\}$
$\mathsf{T}_{\lambda,\mathcal{S}} : \mathsf{Exp}^{\mathsf{MCL}} \to \mathsf{Com}$

- $\mathsf{T}_{\lambda,\mathcal{S}}\ (x) = x$
- $\mathsf{T}_{\lambda,(\mathcal{S},\ell)}\ (\mathsf{new}\ c()) = \mathsf{new}\ x(\lambda_{\mathsf{Loader}}(\ell, c), c)()$
- $\mathsf{T}_{\lambda,(\mathcal{S},\ell)}\ \left(c.m({e_i}^{i\in n})\right) = x(\lambda_{\mathsf{Loader}}(\ell, c), c).m({e'_i}^{i\in n})$, with $\mathsf{T}_{\lambda,(\mathcal{S},\ell)}\ (e_i) = e'_i$, $i \in 1..n$
- $\mathsf{T}_{\lambda,(\mathcal{S},\ell)}\ (\mathsf{ret}\ (e)) = \mathsf{ret}\ (e')$, with $\mathsf{T}_{\lambda,\mathcal{S}}\ (e) = e'$
- $\mathsf{T}_{\lambda,\mathcal{S}}\ (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) = \mathsf{let}\ x = e'_1\ \mathsf{in}\ e'_2$, with $\mathsf{T}_{\lambda,\mathcal{S}}\ (e_i) = e'_i$, $i \in \{1, 2\}$

Figure 7: Translation

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

[9] 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.

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