A syntactic model of mutation and aliasing

Paola Giannini
Computer Science Institute, DiSIT, Università del Piemonte Orientale
Italy
paola.giannini@uniupo.it

Marco Servetto
School of Engineering and Computer Science,Victoria University of Wellington
New Zealand
servetto@ecs.vuw.ac.nz

Elena Zucca
DIBRIS, Università di Genova
Italy
elena.zucca@unige.it

Traditionally, semantic models of imperative languages use an auxiliary structure which mimics memory. In this way, ownerships and other encapsulation properties need to be reconstructed from the graph structure of such global memory. We present an alternative *syntactic* model where memory is encoded as part of the program rather than as a separate resource. This means that execution can be modelled by just rewriting source code terms, as in semantic models for functional programs. Formally, this is achieved by the block construct, introducing local variable declarations, which play the role of memory when their initializing expressions have been evaluated. In this way, we obtain a language semantics which is more abstract, and directly represents at the syntactic level constraints on aliasing, allowing simpler reasoning about related properties. We illustrate this advantage by expressing in the calculus the *capsule* property, characterizing an isolated portion of memory, which cannot be reached through external references. Capsules can be safely moved to another location in the memory, without introducing sharing. We state that the syntactic model can be encoded in the conventional one, hence efficiently implemented, and outline the proof that the dynamic semantics are equivalent.

1 Introduction

In an ongoing stream of work [8, 20, 11, 13, 14] on type systems to control sharing and mutation in imperative languages, we have adopted a non-conventional operational model, where memory is encoded as part of the program rather than as a separate resource. In this setting execution can be modelled by just rewriting source code terms, as in semantic models for functional programs.

The advantage of this choice is that language semantics is more abstract, and directly represents at the syntactic level constraints on aliasing, allowing simpler reasoning about related properties.

In this paper, we focus on the operational model itself, rather than on type systems, and formalize its relation with the conventional model, where an auxiliary global structure mimics memory.

To informally introduce this syntactic calculus, we show examples of reduction sequences. The main idea is to use variable declarations to directly represent the memory. That is, a declared variable is not replaced by its value, as in standard `let`, but the association is kept and used when necessary, as it happens, with different aims and technical problems, in cyclic lambda calculi [4, 17, 3]. Assuming a program (class table) where class `C` has two fields `f1` and `f2` of type `D`, and class `D` has a field `f` of type `D`, the term

```
D x=new D(y); D y=new D(x); C w={D z2= new D(z2); y.f=z1; new C(z2,z2)}; w.f1
```

starts with two declarations that can be seen as a memory consisting of two mutually referring objects. Then there is a declarations whose right-hand-side needs to be evaluated and the final expression return the value of a field of the object associated with this last variable. The reduction of the term is as follows

Submitted to:
DCM 2018

```
D x=new D(y); D y=new D(x); C w={D z2= new D(z2); y.f=y; new C(z2,z2)}; w.f1 ⟶
D x=new D(y); D y=new D(y); C w={D z2=new D(z2); new C(z2,z2)}; w.f1 ⟶
D x=new D(y); D y=new D(y); D z2 = new D(z2); C w={new C(z2,z2)}; w.f1 ⟶
D x=new D(y); D y=new D(y); D z2 = new D(z2); C w = new C(z2,z2); w.f1 ⟶
D x=new D(y); D y=new D(y); D z2 = new D(z2); C w = new C(z2,z2); z2 ⟶
D z2 = new D(z2); z2
```

Evaluation proceeds left to right. We start evaluating the right-hand-side of the declaration for `w`, by updating the field `f` of `y`. Then in order to evaluate the field access `w.f1`, two preliminary reductions are needed: the first to move the declaration of `z2` outside of the inner block, and the second to eliminate the block. Now the field access `w.f1` can be performed, getting `z2`, and the last step removes declarations (memory) which are not reachable from `z2`, giving as final result a memory consisting of only one cyclic object.

To illustrate how aliasing constraints are directly represented in this syntactic model, let us consider the *capsule* property, characterizing an isolated portion of memory, which cannot be reached through external references. The capsule property has been widely studied in the literature, under different names and variants, such as *isolated* [15], *unique* [7] and *externally unique* [9], *balloon* [2, 19], *island* [10]. In the syntactic calculus, capsules can be characterized in a very simple way, as block values with no free variables. For instance, from the second line of the above example, we can see that the right-hand side of the declaration of `w` is a closed block value, that is, a capsule. Moreover, differently from the conventional model where values are only references, capsules are *first-class values*, which can be passed, e.g., as arguments to methods. They can be also assigned to *affine* variables, that is, variables used at most once, to be safely moved to another location in the memory, without introducing sharing. For instance, in the above example, variable `w` could be declared affine. In this case starting from line 2, reduction would be different:

```
D x=new D(y); D y=new D(y); C^a w={D z2 = new D(z2); new C(z2,z2)}; w.f1 ⟶
D x=new D(y); D y=new D(y); {D z2 = new D(z2); new C(z2,z2)}.f1 ⟶
D x=new D(y); D y=new D(y); {D z2 = new D(z2); new C(z2,z2).f1} ⟶
D x=new D(y); D y=new D(y); {D z2 = new D(z2); z2} ⟶
D z2 = new D(z2); z2
```

Note that if the initial term would be, instead:

```
D x=new D(y); D y=new D(x); C w={D z2= new D(z2); y.f=y; new C(z2,y)}; w.f1
```

then the inner block would *not* reduce to a capsule.

In Sect.2 and Sect.3 we define the conventional and the syntactic calculus, respectively. In Sect.4 we show that the syntactic model can be encoded in the conventional one, hence efficiently implemented, and outline the proof that the dynamic semantics are equivalent. Finally, in Sect.5 we draw some conclusion.

2 The conventional calculus

We illustrate our approach in the context of calculi with an object-oriented flavour, inspired by Featherweight Java [16] (FJ for short). This is only a presentation choice: the ideas and results of the paper could be rephrased in different imperative calculi, e.g., supporting data type constructors and reference types. For the same reason, we omit features such as inheritance and late binding, which are orthogonal to our focus.

The conventional calculus is given in Fig.1. It is similar to other imperative variants of FJ which can be found in the literature [1, 6, 5]. We assume sets of *variables* x, y, z, *class names* C, D, *field names* f, and *method names* m. We adopt the convention that a metavariable which ends in s is implicitly defined as a (possibly empty) sequence in which elements may or may not be separated by commas. In particular, ds (and dvs in the next section) are sequences of d (and dv) and es, vs and xs are sequences of e, v and x separated by commas.

$$
\begin{array}{lcll}
e & ::= & x \mid e.f \mid e.f{=}e' \mid \texttt{new}\ C(es) \mid e.m(es) \mid \{ds\ e\} \mid \iota & \text{expression} \\
d & ::= & Cx{=}e; & \text{declaration} \\
\\
v & ::= & \iota & \text{value} \\
\mathscr{E} & ::= & [\,] \mid \mathscr{E}.f \mid \mathscr{E}.f{=}e' \mid \iota.f{=}\mathscr{E} \mid \texttt{new}\ C(vs,\mathscr{E},es) & \text{evaluation context} \\
 & & \mid \mathscr{E}.m(es) \mid \iota.m(vs,\mathscr{E},es) \mid \{Cx{=}\mathscr{E};\ ds\ e\} &
\end{array}
$$

$$\text{(CTX)}\ \frac{e|\mu \Longrightarrow e'|\mu'}{\mathscr{E}[e]|\mu \Longrightarrow \mathscr{E}[e']|\mu'} \qquad \text{(FIELD-ACCESS)}\ \iota.f_i|\mu \Longrightarrow v_i|\mu \quad \begin{array}{l}\mu(\iota) = \texttt{new}\ C(v_1,\ldots,v_n) \\ \mathsf{fields}(C) = C_1\,f_1 \ldots C_n\,f_n \wedge 1 \le i \le n\end{array}$$

$$\text{(FIELD-ASSIGN)}\ \iota.f_i{=}v|\mu \Longrightarrow v|\mu^{\iota.i=v} \quad \begin{array}{l}\mu(\iota) = \texttt{new}\ C(vs) \\ \mathsf{fields}(C) = C_1\,f_1 \ldots C_n\,f_n \wedge 1 \le i \le n\end{array}$$

$$\text{(NEW)}\ \texttt{new}\ C(vs)|\mu \Longrightarrow \iota|\mu[\texttt{new}\ C(vs)/\iota] \quad \iota \notin \mathsf{dom}(\mu)$$

$$\text{(INVK)}\ \iota.m(v_1,\ldots,v_n)|\mu \Longrightarrow e[\iota/\texttt{this}][v_1/x_1 \ldots v_n/x_n]|\mu \quad \begin{array}{l}\mu(\iota) = \texttt{new}\ C(vs) \\ \mathsf{meth}(C,m) = \langle x_1 \ldots x_n, e\rangle\end{array}$$

$$\text{(DEC)}\ \{Cx{=}\iota;\ ds\ e\}|\mu \Longrightarrow \{ds\ e\}[\iota/x]|\mu \qquad \text{(BLOCK-ELIM)}\ \{\ e\}|\mu \Longrightarrow e|\mu$$

Figure 1: Conventional calculus

An expression can be a variable (including the special variable `this` denoting the receiver in a method body), a field access, a field assignment, a constructor invocation, a method invocation, or a block consisting of a sequence of local variable declarations and a body. In addition, a (runtime) expression can be an *object identifier* ι. Blocks are included to have a more direct correspondence with the syntactic calculus. In a block, a declaration specifies a type (class name), a variable and an initialization expression. We assume that in well-formed blocks there are no multiple declarations for the same variable, that is, *ds* can be seen as a map from variables to expressions.

The class table is abstractly modelled by the following functions:

- $\mathsf{fields}(C)$ gives, for each declared class C, the sequence $C_1\,f_1 \ldots C_n\,f_n$ of its fields declarations.
- $\mathsf{meth}(C,m)$ gives, for each method m declared in class C, the pair consisting of its parameters and body.

The reduction relation $\Longrightarrow$ is defined on pairs $e|\mu$ where a *memory* μ is a finite map from object identifiers ι into object states of shape $\texttt{new}\ C(vs)$. Values are object identifiers (we do not identify the two sets since, extending the language, values would be extended to include, e.g., primitive values).

Evaluation contexts and reduction rules are straightforward. We denote by $\mu^{\iota.i=v}$ the memory where the i-th field of the object state associated to ι has been replaced by v. Local variable declarations have the standard substitution semantics, and are elaborated in the usual left-to-right order (no recursion is allowed). Finally, a block with no declarations is reduced to its body.

3 The syntactic calculus

The syntax of the expressions, given in Fig.2, is the same of the conventional calculus, apart that runtime expressions (object identifiers) are not needed. To lighten the notation, we use the same metavariables.

In the examples, we generally omit the brackets of the outermost block, and abbreviate $\{T\,x{=}e;\ e'\}$ by

$e\,;e'$ when x does not occur free in e'. We also assume to have integer constants, which are not included in the formalization.

e	::=	$x \mid e.f \mid e.f\texttt{=}e' \mid \texttt{new}\ C\texttt{(}es\texttt{)} \mid e.m\texttt{(}es\texttt{)} \mid \{^X ds\ e\}$	expression
d	::=	$T\,x\texttt{=}e\texttt{;}$	declaration
T	::=	C^μ	declaration type
μ	::=	$\varepsilon \mid \texttt{a}$	optional modifier
v	::=	$x \mid \{^X dvs\ x\}$	value
dv	::=	$C\,x\texttt{=new}\ C\texttt{(}xs\texttt{)}\texttt{;}$	evaluated declaration
$\mathscr{E}$	::=	$[\,] \mid \mathscr{E}.f \mid \mathscr{E}.f\texttt{=}e' \mid x.f\texttt{=}\mathscr{E} \mid \texttt{new}\ C\texttt{(}xs,\mathscr{E},es\texttt{)} \mid \mathscr{E}.m\texttt{(}es\texttt{)} \mid x.m\texttt{(}vs,\mathscr{E},es\texttt{)} \mid \mathscr{E}_b$	evaluation context
$\mathscr{E}_b$	::=	$\{^X dvs\ C\,y\texttt{=}\mathscr{E}\texttt{;}\ ds\ e\} \mid \{^X dvs\ \mathscr{E}\}$	block context
$\mathscr{E}_v$	::=	$[\,].f \mid [\,].f\texttt{=}e' \mid x.f\texttt{=}[\,] \mid \texttt{new}\ C\texttt{(}xs,[\,],es'\texttt{)} \mid [\,].m\texttt{(}es\texttt{)}$	value context

Figure 2: Syntactic calculus: syntax, values, and evaluation contexts

Moreover, some annotations are inserted in terms. Namely:

- Local variable declarations (and method parameters) are possibly annotated with a modifier `a`, which, if present, indicates that the variable is *affine*. An affine variable can occur at most once in its scope, and should be initialized with a *capsule*, that is, an isolated portion of store. In this way, it can be used as a temporary reference, to "move" a capsule to another location in the store, without introducing sharing.
- Blocks are annotated with a set X of variables, assumed to be a subset of the locally declared variables. During reduction, if a block $\{^X ds\ e\}$ should reduce to a capsule, only declarations of variables which are not in X can be safely moved outside of the block, see rule (MOVE-DEC). In this paper, since our focus is on the operational model, we do not care about how block annotations are generated. Of course, a trivial overapproximation consists in taking as X the set of all declared variables; a better approximation is taking only those which are used (that is, have some free occurrence in initialization expressions/body), or, even better, are transitively used by the body, in the sense formally defined below. We have shown in previous work [13, 12, 14] that through a type and effect system it is possible to obtain the set of only the declared variables which *will be possibly connected with the final result of the block*.

A sequence *dvs* of *evaluated declarations* plays the role of the memory in the conventional calculus, that is, each *dv* can be seen as an association of an *object state* $\texttt{new}\ C\texttt{(}xs\texttt{)}$ to a reference.

A value is either a variable (a reference to an object), or a block where the declarations are evaluated (hence, correspond to a local memory).
We assume that, in a block value $\{^X dvs\ x\}$, $dvs \neq \varepsilon$ and $dvs_{|x} = dvs$, where, given a sequence of declarations $ds \equiv T_1\,x_1\texttt{=}e_1\texttt{;}\ldots T_n\,x_n\texttt{=}e_n\texttt{;}$ and an expression e, $ds_{|e}$ are the declarations of variables (transitively) used by e, that is, free either in e or in some e_i such that x_i is transitively used by e.

In the syntactic calculus, capsules can be characterized in a very simple way: indeed, a value is a capsule, written $\texttt{caps}(v)$, if it is a closed block value, that is, of shape $\{dvs\ x\}$ with no free variables. The above requirement that all local variables must be transitively used by x is needed, indeed, since otherwise a block value containing "useless" free variables, e.g., $\{C\,x\texttt{=new}\ C\texttt{()}\texttt{;}\ D\,y\texttt{=new}\ D\texttt{(}z\texttt{)}\texttt{;}\ x\}$ would be not recognized to be a capsule. Useless evaluated declarations are removed by rule (GARBAGE).

Evaluation contexts $\mathscr{E}$ are mostly standard. Note that values are assumed to be references, apart from arguments of method calls, which are allowed to be block values. This models the fact that block values (hence, capsules) are first-class values which can be passed to methods. However, they need to

be "opened" when we perform an actual operation on them. We distinguish two subsets of evaluation contexts which will play a special role in the reduction rules. A *block context* $\mathscr{E}_b$ is an evaluation context with the shape of a block. We denote by $\mathsf{get}(\mathscr{E}_b, x)$ the object state associated to x in dvs, if any, and by $\mathsf{inner}(\mathscr{E}_b)$ the variables declared in inner blocks, that is, the hole binders of the direct subcontext $\mathscr{E}$ (the standard formal definition is omitted).

A *value context* $\mathscr{E}_v$ is an evaluation context with the shape of either a field access, or a field assignment, or a constructor invocation, or a method invocation, where the hole (expected to be filled with a block value) is a direct subterm (the receiver in the last case).

We write $\mathsf{FV}(ds)$ and $\mathsf{FV}(e)$ for the free variables of a sequence of declarations and of an expression, respectively, and $X[y/x]$, $ds[y/x]$, and $e[y/x]$ for the capture-avoiding variable substitution on a set of variables, a sequence of declarations, and an expression, respectively, all defined in the standard way.

Expressions are identified modulo congruence, denoted by $\cong$, defined as the smallest congruence satisfying the axioms in Fig.3. Rule (ALPHA) is the usual α-conversion. The condition $x, y \notin \mathsf{dom}(ds\, ds')$ is implicit by well-formedness of blocks. Rule (REORDER) states that we can move evaluated declarations in an arbitrary order. Note that, instead, ds and ds' cannot be swapped, because this could change the order of side effects. In rule (NEW), a constructor invocation can be seen as an elementary block where a new object is allocated.

(ALPHA) $\{^{X} ds\ C\,x{=}e;\ ds'\ e'\} \cong \{^{X[y/x]} ds[y/x]\ C\,y{=}e[y/x];\ ds'[y/x]\ e'[y/x]\}$

(REORDER) $\{^{X} ds\ dv\ ds'\ e\} \cong \{^{X} dv\ ds\ ds'\ e\}$ (NEW) $\texttt{new}\ C(es) \cong \{^{\{x\}} C\,x{=}\texttt{new}\ C(es);\ x\}$

Figure 3: Syntactic calculus: congruence rules

Reduction rules are given in Fig.4.

Rule (CTX) is the usual contextual closure.

In rule (FIELD-ACCESS), a field access of shape $x.f$ is evaluated in the block context containing the first enclosing (evaluated) declaration for x, as expressed by the first side condition and the definition of $\mathsf{get}(\mathscr{E}_b, x)$. The fields of the class C of x are retrieved from the class table. If f is the name of a field of C, say, the i-th, then the field access is reduced to the reference x_i stored in this field. The condition $x_i \notin \mathsf{inner}(\mathscr{E}_b)$ ensures that there are no inner declarations for x_i (otherwise x_i would be erroneously bound). This can always be obtained by rule (ALPHA) of Fig.3. For instance, assuming a class table where class `A` has an `int` field, and class `B` has an `A` field `f`, without this side condition, the term:

```
A a= new A(0); B b= new B(a); {A a= new A(1); b.f}
```

would reduce to

```
A a= new A(0); B b= new B(a); {A a= new A(1); a}
```

whereas this reduction is forbidden, and by rule (ALPHA) the term is instead reduced to

```
A a= new A(0); B b= new B(a); {A a1= new A(1); a}
```

In rule (FIELD-ASSIGN), a field assignment of shape $x.f{=}y$ is evaluated in the block context containing the first enclosing (evaluated) declaration for x, as expressed by the first side condition. The fields of the class C of x are retrieved from the class table. If f is the name of a field of C, say, the i-th, then this first enclosing declaration is updated, by replacing the i-th constructor argument by y obtaining the declaration $C\,x{=}\texttt{new}\ C(x_1, x_{i-1}, y, x_{i+1}, \ldots, x_n)$; as expressed by the notation $\mathscr{E}_b^{x.i=y}$ (whose obvious formal definition is omitted). Analogously to rule (FIELD-ACCESS), we have the side condition that $y \notin \mathsf{inner}(\mathscr{E}_b)$. This side condition, requiring that there are no inner declarations for y, prevents scope extrusion, since if

$$\text{(CTX)}\ \frac{e \longrightarrow e'}{\mathscr{E}[e] \longrightarrow \mathscr{E}[e']} \qquad \text{(FIELD-ACCESS)}\ \mathscr{E}_b[x.f_i] \longrightarrow \mathscr{E}_b[x_i] \quad \begin{array}{l}\mathsf{get}(\mathscr{E}_b,x) = \texttt{new}\ C(x_1,\ldots,x_n) \wedge x \notin \mathsf{inner}(\mathscr{E}_b)\\ \mathsf{fields}(C) = C_1\,f_1 \ldots C_n\,f_n \wedge 1 \le i \le n\\ x_i \notin \mathsf{inner}(\mathscr{E}_b)\end{array}$$

$$\text{(FIELD-ASSIGN)}\ \mathscr{E}_b[x.f_i{=}y] \longrightarrow \mathscr{E}_b^{x.i=y}[y] \quad \begin{array}{l}\mathsf{get}(\mathscr{E}_b,x) = \texttt{new}\ C(xs) \wedge x \notin \mathsf{inner}(\mathscr{E}_b)\\ \mathsf{fields}(C) = C_1\,f_1 \ldots C_n\,f_n \wedge 1 \le i \le n\\ y \notin \mathsf{inner}(\mathscr{E}_b)\end{array}$$

$$\text{(INVK)}\ \mathscr{E}_b[x.m(v_1,..,v_n)] \longrightarrow \mathscr{E}[\{C\,\texttt{this}{=}x;\, C_1^{\mu_1}\,x_1{=}v_1; \ldots C_n^{\mu_n}\,x_n{=}v_n;\ e\}] \quad \begin{array}{l}\mathsf{get}(\mathscr{E}_b,x) = \texttt{new}\ C(xs) \wedge x \notin \mathsf{inner}(\mathscr{E}_b)\\ \mathsf{meth}(C,m){=}\langle C_1^{\mu_1}\,x_1..C_n^{\mu_n}\,x_n, e\rangle\end{array}$$

$$\text{(ALIAS-ELIM)}\ \{^X dvs\ C\,x{=}y;\ ds\ e\} \longrightarrow \{^{X\setminus\{x\}} dvs\ ds[y/x]\ e[y/x]\}$$

$$\text{(AFFINE-ELIM)}\ \{^X dvs\ C^{\mathsf{a}}\,x{=}v;\ ds\ e\} \longrightarrow \{^{X\setminus\{x\}} dvs\ ds[v/x]\ e[v/x]\} \quad \mathsf{caps}(v)$$

$$\text{(BLOCK-ELIM)}\ \{^{\emptyset}\ e\} \longrightarrow e \qquad \text{(GARBAGE)}\ \{^X dvs\ ds\ e\} \longrightarrow \{^{X\setminus \mathsf{dom}(dvs)} ds\ e\} \quad (\mathsf{FV}(ds) \cup \mathsf{FV}(e)) \cap \mathsf{dom}(dvs) = \emptyset$$

$$\text{(MOVE-DEC)}\ \{^Y dvs\ C^{\mu}\,x{=}\{^X dvs'\ ds\ e\};\ ds'\ e'\} \longrightarrow \{^Y dvs\ dvs'\ C^{\mu}\,x{=}\{^X ds\ e\};\ ds'\ e'\} \quad \begin{array}{l}\mathsf{FV}(dvs') \cap \mathsf{dom}(ds) = \emptyset\\ \mathsf{FV}(dvs\ ds'\ e') \cap \mathsf{dom}(dvs') = \emptyset\\ \mu = \mathsf{a} \Rightarrow \mathsf{dom}(dvs') \cap X = \emptyset\end{array}$$

$$\text{(MOVE-BODY)}\ \{^Y dvs\ \{^X dvs'\ ds\ e\}\} \longrightarrow \{^Y dvs\ dvs'\ \{^X ds_2\ e\}\} \quad \begin{array}{l}\mathsf{FV}(dvs') \cap \mathsf{dom}(ds) = \emptyset\\ \mathsf{FV}(dvs) \cap \mathsf{dom}(dvs') = \emptyset\end{array}$$

$$\text{(MOVE-SUBTERM)}\ \mathscr{E}_v[\{^X dvs\ dvs'\ v\}] \longrightarrow \{^{X \cap \mathsf{dom}(dvs)} dvs\ \mathscr{E}_v[\{^{X\setminus \mathsf{dom}(dvs)} dvs'\ v\}]\} \quad \begin{array}{l}\mathsf{FV}(dvs) \cap \mathsf{dom}(dvs') = \emptyset\\ \mathsf{FV}(\mathscr{E}_v) \cap \mathsf{dom}(dvs) = \emptyset\end{array}$$

Figure 4: Syntactic calculus: reduction rules

$y \in \mathsf{inner}(\mathscr{E}_b)$, $\mathscr{E}_b^{x.i=y}$ would take y outside the scope of its definition. For example, without this side condition, the term

```
A a= new A(0); B b= new B(a); {A a1= new A(1); b.f=a1}
```

would reduce to

```
A a= new A(0); B b= new B(a1); {A a1= new A(1); a1}
```

which is not correct since `a1` is a free variable. The rules (MOVE-DEC) and (MOVE-BODY) (see below) can be used to move the declaration of y outside its declaration block. So the term reduces, instead, to

```
A a= new A(0); B b= new B(a); A a1= new A(1); b.f=a1
```

by applying first rule (MOVE-BODY), and then (BLOCK-ELIM). Now the term correctly reduces to

```
A a= new A(0); B b= new B(a1); A a1= new A(1); a1
```

In rule (INVK), a method call of shape $x.m(v_1,..,v_n)$ is evaluated in the block context containing the first enclosing (evaluated) declaration for x, as expressed by the first side condition. Method m of C, if any, is retrieved from the class table. The call is reduced to a block where declarations of the appropriate type for `this` and the parameters are initialized with the receiver and the arguments, respectively, and the body is the method body. If a parameter is affine, then the corresponding argument will be checked to be a capsule when the formal parameter will be substituted with the associated value.
The following two rules eliminate declarations from a block.

In rule (ALIAS-ELIM), a reference (non affine variable) x which is initialized as an alias of another reference y is eliminated by replacing all its occurrences. In rule (AFFINE-ELIM), an affine variable is eliminated by replacing its unique occurrence with the value (required to be a capsule) associated to its declaration.
By rule (BLOCK-ELIM), a block with no declarations is reduced to its body. Rule (GARBAGE) states that we can remove a useless sequence of evaluated declarations from a block. Note that it is only possible to safely remove declarations which are evaluated, since they do not have side effects.
With the remaining rules we can move a sequence of evaluated declarations from a block to the directly enclosing block, as it happens with rules for *scope extension* in the π-calculus [18].
In rules (MOVE-DEC) and (MOVE-BODY), the inner block is the right-hand side of a declaration, or the body, respectively, of the enclosing block. The first two side conditions ensure that moving the declarations dvs' does cause neither scope extrusion nor capture of free variables. More precisely: the first prevents moving outside a declaration dvs' which depends on local variables of the inner block. The second prevents capturing with dvs' free variables of the enclosing block. Note that the second condition can be obtained by α-conversion of the inner block, but the first cannot. Finally, when the block initializes an affine variable, the third side condition of rule (DEC) forbids to move outside the block declarations of variables that will be possibly connected to the result of the block. This is because the block should ultimately reduce to a closed expression.

In case of a non affine declaration, instead, this is not a problem.
Rule (MOVE-SUBTERM) handles the cases when the inner block is a subterm of a field access, field assignment, constructor invocation, or method invocation. Note that in this case the inner block is necessarily a (block) value. We use value contexts to express all such cases in a compact way.

4 Preservation of semantics

In this section, for clarity, we use $\hat{e}$ and $\widehat{ds}$ to range over expressions and sequences of declarations of the conventional calculus, which could include object identifiers.

We show that, if an expression has a reduction sequence in the syntactic calculus, then it has an "equivalent" reduction sequence in the conventional calculus. Note that the converse does not hold, since an expression which is stuck in the syntactic calculus (since a capsule runtime check fails) could reduce in the conventional calculus.

In order to state the preservation theorem, we define a relation $\overset{\rho}{\hookrightarrow}$ between expressions e of the syntactic calculus and pairs $\hat{e}|\mu$ of the conventional calculus. The relation is labelled by an environment ρ which maps variables[1] into object identifiers. Intuitively, $e \overset{\rho}{\hookrightarrow} \hat{e}|\mu$ holds if the evaluated declarations in e have been encoded in the memory μ through ρ. More precisely, for all declarations of the shape $C\,x$=new $C(xs)$; occurring in e, the environment ρ is defined on x and on all elements xs. Moreover, $\mu(\rho(x)) = \texttt{new}\ C(\rho(xs))$, and $\hat{e}$ is obtained from e by removing such evaluated declarations and applying the substitution ρ. Formally:

$x \overset{\rho}{\hookrightarrow} \iota|\mu$ if $\rho(x) = \iota$
$e.f \overset{\rho}{\hookrightarrow} \hat{e}.f|\mu$ if $e \overset{\rho}{\hookrightarrow} \hat{e}|\mu$
(other analogous propagation rules are omitted)
$\{^X dvs\ ds\ e\} \overset{\rho}{\hookrightarrow} \{\widehat{ds}\ \hat{e}\}$ if:
$dvs = C_1\,x_1$=new $C_1(xs_1)$;$\ldots C_n\,x_n$=new $C_n(xs_n)$; $\quad ds = D_1\,y_1$=e_1;$\ldots D_m\,y_m$=e_m; $\quad \widehat{ds} = D_1\,y_1$=$\hat{e}_1$;$\ldots D_m\,y_m$=$\hat{e}_m$;
$e_i \overset{\rho}{\hookrightarrow} \hat{e}_i|\mu$, for $i \in 1..m$, $e \overset{\rho}{\hookrightarrow} \hat{e}|\mu$, and $\mu(\rho(x_i)) = \texttt{new}\ C_i(\rho(xs_i))$, for $i \in 1..n$.

Theorem 1. *If $e \longrightarrow^\star v$, and $e \overset{\rho}{\hookrightarrow} \hat{e}|\mu$, then $\hat{e}|\mu \Longrightarrow^\star \iota|\mu'$ with $v \overset{\rho'}{\hookrightarrow} \iota|\mu'$ for some ρ', μ' such that $\rho \subseteq \rho'$,*

[1] Assuming to obtain no shadowing by α-renaming.

$dom(\mu) \subseteq dom(\mu')$.

Proof. By arithmetic induction on the number of steps.

Base If $e \longrightarrow^0 v$, then $e \equiv v \equiv \{^X C_1\, x_1\texttt{=new}\ C_1\texttt{(}xs_1\texttt{)};\ldots C_n\, x_n\texttt{=new}\ C_n\texttt{(}xs_n\texttt{)};\ x\}$. From the definition of $\overset{\rho}{\hookrightarrow}$ we have that $\hat{e} = \{\ \iota\}$, $\rho(x) = \iota$, and $\mu(\rho(x_i)) = \texttt{new}\ C_i\texttt{(}\rho(xs_i)\texttt{)}$, for $i \in 1..n$. Hence, $\{\ \iota\}|\mu$ reduces in one step, by rule (BLOCK-ELIM), to $\iota|\mu$, and $v \overset{\rho}{\hookrightarrow} \iota|\mu$.

Inductive step If $e \longrightarrow^{n+1} v$, then $e \longrightarrow e'$ and $e' \longrightarrow^n v$. By inductive hypothesis we have that, if $e' \overset{\rho'}{\hookrightarrow} \hat{e}'|\mu'$, then $\hat{e}'|\mu' \Longrightarrow^\star \iota|\mu''$ and $v \overset{\rho''}{\hookrightarrow} \iota|\mu''$, for some ρ'', μ'' such that $\rho' \subseteq \rho''$, $\mathsf{dom}(\mu') \subseteq \mathsf{dom}(\mu'')$. Then, it is enough to show that:

> if $e \overset{\rho}{\hookrightarrow} \hat{e}|\mu$, and $e \longrightarrow e'$, then
> $\hat{e}|\mu \Longrightarrow^\star \hat{e}'|\mu'$ such that $e' \overset{\rho'}{\hookrightarrow} \hat{e}'|\mu'$, for some $\hat{e}'$, ρ', μ' such that $\rho \subseteq \rho'$, $\mathsf{dom}(\mu) \subseteq \mathsf{dom}(\mu')$.

This can be proved by induction on the reduction rules of the syntactic calculus. □

One crucial point for the preservation theorem is that the substitution semantics, modeling "moving" capsules from a location to another in the memory, see rule (AFFINE-ELIM), is indeed equivalent to the conventional semantics. Note that this only holds for variables which are affine, that is, used at most once. As a counterexample consider, for instance, the term

```
Ca c= {C b=new C(0) b}; c.f=3; c.f
```

which would reduce to

```
{C b=new C(0) b}.f=3; {C b=new C(0) b}.f
```

and then in some steps to `0`. Instead with the conventional semantics the term would reduce to `3`.

5 Conclusion

In this paper we presented a calculus for an imperative object oriented language whose distinguished feature are the following

- Local variable declarations are used to directly represent the memory. That is, a declared (non affine) variable is not replaced by its value, as in standard `let`, but the association is kept and used when necessary.
- In this way, there are language values (block values) which represent (a portion of) memory, and the fact that such portion of memory is a capsule can be *modularly* checked by inspecting only the value itself, without any need to explore the whole graph structure of the global memory as it would be in the conventional model.
- To safely handle capsules, the syntactic calculus supports *affine* variables with a special semantics and block annotations. Runtime checks ensure that their initializing value is a capsule (side condition in rules (AFFINE-ELIM)), and their (unique by definition) occurrence is replaced by their capsule value (rule (AFFINE-ELIM)).

In previous work [8, 20, 11, 13, 14], we have designed type systems which statically ensure that such runtime checks succeed, hence execution is not stuck. In this paper we outlined a proof that the dynamic semantics of the calculus coincides with the standard semantics of imperative calculi relying on a global memory. In such calculi reasoning about program properties such as sharing requires the formalization

of invariants on the memory and the proof of their preservation under reduction, whereas in ours this can be done by structural induction on terms. In future work we plan to complete the proof of preservation and add the modelling of immutable references. We will also investigate (a form of) Hoare logic on top of our model.

References

[1] Alexander Joseph Ahern & Nobuko Yoshida (2005): *Formalising Java RMI with explicit code mobility*. *OOPSLA 2005*, ACM Press, pp. 403–422,

[2] Paulo Sérgio Almeida (1997): *Balloon Types: Controlling Sharing of State in Data Types*. In: *ECOOP*, *LNCS* 1241, Springer, pp. 32–59.

[3] Zena M. Ariola & Stefan Blom (2002): *Skew confluence and the lambda calculus with letrec*. *Ann. Pure Appl. Logic* 117(1-3), pp. 95–168.

[4] Zena M. Ariola & Matthias Felleisen (1997): *The Call-by-Need Lambda Calculus*. *Journ. of Functional Programming* 7(3), pp. 265–301.

[5] Lorenzo Bettini, Ferruccio Damiani & Ina Schäfer (2010): *IFJ: a minimal imperative variant of FJ*. TTR 133/2010, Dipartimento di Informatica, Università di Torino.

[6] Gavin M. Bierman & Matthew J. Parkinson (2003): *Effects and effect inference for a core Java calculus*. *ENTCS* 82(7), pp. 82–107.

[7] John Boyland (2010): *Semantics of Fractional Permissions with Nesting*. *ACM TOPLAS* 32(6).

[8] Andrea Capriccioli, Marco Servetto & Elena Zucca (2016): *An imperative pure calculus*. *ENTCS* 322, pp. 87–102.

[9] David Clarke & Tobias Wrigstad (2003): *External Uniqueness is Unique Enough*. *ECOOP* , *LNCS* 2473, Springer, pp. 176–200.

[10] Werner Dietl, Sophia Drossopoulou & Peter Müller (2007): *Generic Universe Types*. *ECOOP*, *LNCS* 4609, Springer, pp. 28–53.

[11] Paola Giannini, Marco Servetto & Elena Zucca (2016): *Types for Immutability and Aliasing Control*. *ICTCS* , *CEUR Workshop Proceedings* 1720, CEUR-WS.org, pp. 62–74.

[12] Paola Giannini, Marco Servetto & Elena Zucca (2017): *Tracing sharing in an imperative pure calculus: extended abstract*. *FTfJP*, ACM Press, pp. 6:1–6:6.

[13] Paola Giannini, Marco Servetto & Elena Zucca (2017): *A type and effect system for sharing*. *OOPS*, ACM Press, pp. 1513–1515.

[14] Paola Giannini, Marco Servetto & Elena Zucca (2018): *A type and effect system for uniqueness and immutability*. *OOPS*, ACM Press. To appear.

[15] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield & Joe Duffy (2012): *Uniqueness and reference immutability for safe parallelism*. *OOPSLA* , ACM Press, pp. 21–40.

[16] Atsushi Igarashi, Benjamin C. Pierce & Philip Wadler (2001): *Featherweight Java: a minimal core calculus for Java and GJ*. *ACM TOPLAS* 23(3), pp. 396–450.

[17] John Maraist, Martin Odersky & Philip Wadler (1998): *The Call-by-Need Lambda Calculus*. *Journ. of Functional Programming* 8(3), pp. 275–317.

[18] Robin Milner (1999): *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press.

[19] Marco Servetto, David J. Pearce, Lindsay Groves & Alex Potanin (2014): *Balloon Types for Safe Parallelisation over Arbitrary Object Graphs*. *WODET*.

[20] Marco Servetto & Elena Zucca (2015): *Aliasing Control in an Imperative Pure Calculus*. *APLAS*, *LNCS* 9458, Springer, pp. 208–228.