An imperative pure calculus[1]

Andrea Capriccioli

DIBRIS
Università di Genova
Italy

Marco Servetto[2]

Victoria University of Wellington
New Zealand

Elena Zucca[3]

DIBRIS
Università di Genova
Italy

Abstract

We present a simple calculus where imperative features are modeled by just rewriting source code terms, rather than by modifying an auxiliary structure which mimics physical memory. Formally, this is achieved by the block construct, introducing local variable declarations, which also plays the role of store when such declarations 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 possibility by a simple extension of the standard type system which assigns a `capsule` tag to expressions that will reduce to (values representing) isolated portions of store.

1 Introduction

Traditional execution models for imperative languages use an auxiliary structure, called *memory* or *store*, which is a mathematical abstraction of the physical memory, and is typically a map from *locations* (modeling memory addresses) into storable values. Locations are a runtime notion, and their names are globally available, that is, memory is flat, whereas *variables* are a language notion, and their names obey scoping rules (shadowing) and α-conversion.

[1] This work has been partially supported by MIUR CINA - Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.

[2] Email: marco.servetto@ecs.vuw.ac.nz

[3] Email: elena.zucca@unige.it

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

In this paper, we propose an alternative, more abstract, execution model which is a *pure calculus*. That is, execution is modeled by just rewriting source code terms, in the same way lambda calculus models functional languages.

The following is an example of reduction sequence in the calculus, where we emphasize at each step the redex which is reduced.

```
D z=new D(0)   C x=new C(z,z)   [ C y=x ]  D w=new D(y.f1.f+1) x.f2=w  x ⟶
D z=new D(0)   C x=new C(z,z)   D w=new D( [ x.f1 ].f+1) x.f2=w x ⟶
D z=new D(0)   C x=new C(z,z)   D w=new D( [ z.f ]+1) x.f2=w  x ⟶
D z=new D(0)   C x=new C(z,z)   D w=new D( [ 0+1 ]) x.f2=w  x ⟶
D z=new D(0)   C x=new C(z,z)   D w=new D(1)    [ x.f2=w ] x ⟶
D z=new D(0)   C x=new C(z,w)   D w=new D(1)    x
```

The main idea is to use local variable declarations, as in the `let` construct, to directly represent 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. [4]

The calculus is designed with an object-oriented flavour [5], inspired to Featherweight Java [14]. That is, assuming a program (class table) where class `C` has two fields `f1` and `f2` of type `D`, and class `D` has an integer field `f`, in the initial term the first two declarations can be seen as a store which associates to `z` an object of class `D` whose field contains `0`, and to `x` an object of class `C` whose two fields contains (a reference to) the previous object. The first reduction step eliminates an alias, by replacing occurrences of `y` by `x`. The next three reduction steps compute `x.f1.f+1`, by performing two field accesses and one sum. The last step performs a field assignment. The final result of the evaluation is an object of class `C` whose fields contain two objects of class `D`, whose field contains `0` and `1` field, respectively. As usual, references in the store can be mutually recursive, as in the following example, where we assume a class `B` with a field of type `B`.

```
B x= new B(y)   B y= new B(x)   y
```

In the examples until now, memory is flat, as it usually happens in models of imperative languages. However, in our calculus, we are also able to represent a hierarchical memory, as shown in the example below, where we assume a class `A` with two fields of type `B` and `D`, respectively.

```
D z= new D(0)
A w= ( B x= new B(y)   B y= new B(x)   A u= new A(x,z)   u)
w
```

Here, the store associates to `w` a block introducing local declarations, that is, in turn a store. [6] The advantage of this representation is that it models in a simple and natural way constraints about aliasing among objects, notably:

- the fact that an object is not referenced from outside some enclosing object is

[4] As it happens, with different aims and technical problems, in cyclic lambda calculi [4,3], see the Conclusion for more comments.

[5] This is only a presentation choice: all the ideas and results of the paper could be easily rephrased, e.g., in a ML-like syntax with data type constructors and reference types.

[6] In the examples, we omit for readability the brackets of the outermost block.

directly modeled by the block construct: for instance, the object denoted by `y` can only be reached through `w`

- conversely, the fact that an object does not refer to the outside is modeled by the fact that the corresponding block is closed (that is, has no free variables): for instance, the object denoted by `w` is not closed, since it refers to the external object `z`.

Note that both information is kept also in the following term

```
D z= new D(0)   B x= new B(y)   B y= new B(x)   A u= new A(x,z)   u
```

but should be reconstructed by computing dependencies among variables. In other words, our calculus smoothly integrates memory representation with shadowing and α-conversion. However, there is a problem which needs to be handled to keep this representation correct: reading (or, symmetrically, updating) a field could cause scope extrusion. For instance, the term `C y= ( D z= new D(0) C x= new D(z,z) x) y.f` would reduce to `C y= ( D z= new D(0) C x= new D(z,z) x) z`. To avoid this problem, the above reduction step is forbidden. However, reduction is not stuck, since we can transform the above term in the equivalent term where the inner block has been flattened, and get the following correct reduction sequence:

```
D z= new D(0)   C x= new D(z,z)   C y= x   y.f ⟶
D z= new D(0)   C x= new D(z,z)   x.f ⟶
D z= new D(0)   z
```

That is, we consider expressions to be equivalent modulo moving a sequence of declarations from a block to the directly enclosing block, and conversely, and this equivalence is used exactly in the same way α-equivalence is used in lambda calculus, to allow reduction steps which would, otherwise, be prevented since not correct. Note also that in the final term the declaration of `x` has been be removed (more precisely, we get this simplified term again by equivalence) since useless.

An imperative calculus without store has been preliminarly proposed in [17], where, however, reduction rules required a stack of sequences of local declarations as auxiliary structure. In this paper, we formalize the same idea by a pure calculus, where only language terms are reduced, providing a simple and natural foundational model for imperative languages, analogous, as said above, to lambda calculus for functional languages.

Besides its elegance and simplicity, this language execution model is not driven by the machine implementation and does not rely on runtime structures that do not exist in the source program. More importantly, it can constitute the basis for many important research directions, since, as illustrated above, object graph topologies are directly formalized in syntactic way, hence their properties can be expressed and formally verified more naturally and easily. Even though the focus of the current paper is on the calculus in itself, in order to illustrate these possibilities we provide a simple extension of the standard type system for the language where it is possible to assign to an expression a `capsule` tag, meaning that the expression will reduce to a reachable object subgraph which cannot be aliased from the outside. This notion has many variants in the literature about aliasing (*externally unique* [5], *balloon* [2,19], *island* [13,8], *isolated* [12]).

The rest of the paper is organized as follows: in Section 2 we provide the formal

$$
\begin{array}{lll}
e & ::= x \mid e.f \mid e.m(es) \mid e.f = e' \mid \texttt{new}\ C(es) \mid (ds\ e) & \text{expression} \\
d & ::= Cx = e & \text{declaration} \\
dv & ::= Cx = rv & \text{evaluated declaration} \\
rv & ::= \texttt{new}\ C(xs) \mid (dvs\ v) & \text{right value} \\
v & ::= x \mid rv & \text{value (object)} \\
\mathcal{E} & ::= [\,] \mid \mathcal{E}.f \mid \mathcal{E}.m(es) \mid x.m(xs, \mathcal{E}, es) \mid \mathcal{E}.f = e' \mid x.f = \mathcal{E} & \text{evaluation context} \\
 & \quad \mid \texttt{new}\ C(xs, \mathcal{E}, es) \mid (dvs\ Cx = \mathcal{E}\ ds\ e) \mid (dvs\ \mathcal{E}) &
\end{array}
$$

Fig. 1. Expressions, values, and evaluation contexts

definition of the calculus, in Section 3 the type system, in Section 4 the results, and in Section 5 some conclusion and pointer to further work. The Appendix provides auxiliary definitions. Proofs omitted for lack of space will be provided in a forthcoming extended version of this paper.

2 Calculus

The syntax is given in Figure 1. We assume sets of *variables* x, y, z, ..., *class names* C, *field names* f, and *method names* m. We adopt the convention that a metavariable which ends by s is implicitly defined as a (possibly empty) sequence, for example, ds is defined by $ds ::= \epsilon \mid d\ ds$, where ϵ denotes the empty string.

An expression can be a variable (including the special variable `this` denoting the receiver in a method body), a field access, a method invocation, a field assignment, a constructor invocation and a block consisting of a sequence of declarations and a body. A declaration specifies a type, 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 into expressions, and we use the notation $\mathsf{dom}(ds)$ and $ds(x)$. Moreover, for simplicity, we allow mutual recursion only among evaluated declarations [7], e.g., `(C x= new C(x) x)` is allowed, whereas `(C x= x.f x)` is not. Allowing general recursion would require a sophisticated type system [8], as in [18], but this is not the focus of this paper.

In the examples we feel free to also use expressions of primitive types such as `int`, but they are omitted in the formal definition for simplicity. Moreover, we generally omit the outermost brackets of a block, and abbreviate $(Cx = e\ e')$ by $e\ e'$ when x does not occur free in e'.

A sequence dvs of *evaluated declarations* plays the role of the store in conventional models of imperative languages, that is, each dv can be seen as an association of a *right value* to a variable. Right values can be either *object states*, of shape $\texttt{new}\ C(xs)$, or block values, that is, blocks where all declarations have been evaluated, and the body is (recursively) a value. The latter case allows the store to be hierarchical.

[7] Defined by the third production, and informally explained below.
[8] To avoid access to objects not initialized yet as in the example.

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

(REORDER) $\dfrac{}{(ds\ C\,x = rv\ ds'\ e) \cong (C\,x = rv\ ds\ ds'\ \ e)}$

(GARBAGE) $\dfrac{}{(dvs\ ds\ e) \cong (ds\ e)}\ \mathsf{FV}((ds\ e)) \cap \mathsf{dom}(dvs) = \emptyset$

(ELIM) $\dfrac{}{(\ e) \cong e}$ (NEW) $\dfrac{}{\texttt{new}\ C(es) \cong (C\,x\ \texttt{=new}\ C(es)\ x)}$

(BODY) $\dfrac{}{(ds\ (ds_1\ \ ds_2\ \ e)) \cong (ds\ ds_1\ (ds_2\ \ e))}\ \begin{array}{l}\mathsf{FV}(ds_1) \cap \mathsf{dom}(ds_2) = \emptyset \\ \mathsf{FV}(ds) \cap \mathsf{dom}(ds_1) = \emptyset\end{array}$

(DEC-RIGHT) $\dfrac{}{(ds\ C\,x = (ds_1\ \ ds_2\ \ e)\ \ ds'\ \ e') \cong (ds\ ds_1\ \ C\,x = (ds_2\ \ e)\ \ ds'\ \ e')}\ \begin{array}{l}\mathsf{FV}(ds_1) \cap \mathsf{dom}(ds_2) = \emptyset \\ \mathsf{FV}(ds\,ds') \cap \mathsf{dom}(ds_1) = \emptyset\end{array}$

(FIELD-ACCESS-RCV) $\dfrac{}{(ds\ e).f \cong (ds\ e.f)}$

(METH-CALL-RCV) $\dfrac{}{(ds\ e).m(es) \cong (ds\ e.m(es))}\ \mathsf{FV}(es) \cap \mathsf{dom}(ds) = \emptyset$

(METH-CALL-ARG) $\dfrac{}{e.m(es, (dvs\ e'), es') \cong (dvs\ e.m(es, e', es'))}\ \mathsf{FV}(e, es, es') \cap \mathsf{dom}(dvs) = \emptyset$

(FIELD-ASSIGN-LEFT) $\dfrac{}{(ds\ e).f = e' \cong (ds\ e.f = e')}\ \mathsf{FV}(e') \cap \mathsf{dom}(ds) = \emptyset$

(FIELD-ASSIGN-RIGHT) $\dfrac{}{e.f = (dvs\ e') \cong (dvs\ e.f = e')}\ \mathsf{FV}(e) \cap \mathsf{dom}(dvs) = \emptyset$

(NEW-ARG) $\dfrac{}{\texttt{new}\ C(es, (dvs\ e), es') \cong (dvs\ \texttt{new}\ C(es, e, es'))}\ \mathsf{FV}(es, es') \cap \mathsf{dom}(dvs) = \emptyset$

Fig. 2. Congruence rules

An object state $\texttt{new}\ C(xs)$ represents an elementary allocation unit, and can be considered as a shorter form for a block $(C\,x\ \texttt{=new}\ C(xs)\ x)$, as formalized by congruence rule (NEW) in Figure 2. Hence, a block value has shape $(dvs_1\ (\ldots\ (dvs_n\ x)\ldots))$, for $n \geq 0$. We call x the *root* of the value, and we assume that in well-formed block values it is bound in some dvs_i.

A value is the final result of the reduction of an expression, and is either a variable (a reference to an object), or an object state, or a block value. A closed expression is expected to reduce to a closed value.

Evaluation contexts express standard left-to-right evaluation. Note that in field access, method invocation, and field assignment subterms are considered evaluated (hence the corresponding action can be performed), when they are variables (references to objects).

We write $\mathsf{FV}(e)$ and $\mathsf{FV}(ds)$ for the free variables of an expression and a sequence of declarations, respectively, formally defined in the Appendix.

Semantics is defined by a *congruence* relation, which captures structural equivalence, and a *reduction* relation, which models actual computation, similarly to what happens, e.g., in π-calculus [15].

The congruence relation, denoted by $\cong$, is defined as the smallest congruence satisfying the axioms given in Figure 2.

Rule (ALPHA) is the usual α-conversion.

By the following two rules we can manipulate the declarations in a block. Rule (REORDER) states that we can move evaluated declarations first, in an arbitrary or-

der. Informally, this is safe since they have no longer side effects. Rule (GARBAGE) states that we can remove (or, conversely, add) a useless sequence of evaluated declarations from a block. Note that it is only possible to safely remove/add declarations which are evaluated, since, otherwise, their evaluation could have side effects.

By the following two rules we can eliminate and introduce blocks. Rule (ELIM) states the obvious fact that a block with no declarations is equivalent to its body. In rule (NEW), a constructor invocation can be seen as an elementary block where a new object is allocated.

By the remaining rules we can move a sequence of declarations from a block to the directly enclosing block, or conversely, as it happens with rules for *scope extension* in the π-calculus [15].

In the first two rules, (BODY) and (DEC-RIGHT), the inner block is the body, or the right-hand side of a declaration, respectively, of the enclosing block. The side conditions ensure that the declarations can be safely moved. More precisely: the former prevents to move outside a declaration which depends on local variables of the inner block. Conversely, the latter prevents to move inside a declaration which is used by other declarations of the enclosing block. Note that both these conditions *cannot* be obtained by α-conversion. Moreover, note that the conditions $\mathsf{dom}(ds_1) \cap \mathsf{dom}(ds_2) = \emptyset$ and $\mathsf{dom}(ds_1) \cap \mathsf{dom}(ds) = \emptyset$ ($\mathsf{dom}(ds_1) \cap \mathsf{dom}(ds\, ds') = \emptyset$ in the second rule) are implicit from well-formedness of blocks.

The other rules handle the cases when the inner block is a direct subterm of a field access, method invocation, field assignment or constructor invocation. In all such cases, the action to be executed is propagated to the body of the block, within the scope of the declarations. Hence, we must avoid capture of free variables, as specified by the side conditions of the rules, which can be always obtained by α-renaming. Moreover, as in rules (REORDER) and (GARBAGE) above, we must preserve the evaluation order, hence in some cases declarations are required to be evaluated, that is, to have no longer side effects.

Reduction rules are given in Figure 3. We write $e[y/x]$ for the expression obtained by replacing all (free) occurrences of x in e by y, and $\mathsf{HB}(\mathcal{E})$ for the *hole binders* of $\mathcal{E}$, that is, the variables declared in blocks enclosing the context hole, both formally defined in the Appendix.

We assume a fixed class table, abstractly modeled by the following functions:

- $\mathsf{fields}(C)$ gives, for each declared class C, the sequence of its fields declarations $C_1\, f_1 \ldots C_n\, f_n$
- $\mathsf{mbody}(C, m)$ gives, for each method m declared in class C, the pair $\langle x_1 \ldots x_n, e\rangle$ consisting of the sequence of its parameters, and its body.

The most interesting reduction rules are those for reading/assigning a field, so we first illustrate these rules in detail, also providing examples, then explain the others.

Field access

In rule (FIELD-ACCESS), given a field access of shape $x.f$, the first enclosing declaration for x is found (side condition $x \notin \mathsf{HB}(\mathcal{E})$ ensures that it is the first), and fields of the class C of x are retrieved from the class table. If f is actually the

$$\text{(CTX)}\quad \frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']} \qquad \text{(CONGR)}\quad \frac{e_1 \longrightarrow e_2 \quad e_1 \cong e_1'}{e_1' \longrightarrow e_2' \quad e_2 \cong e_2'}$$

$$\text{(FIELD-ACCESS)}\quad \frac{}{(dvs\ \mathcal{E}[x.f]) \longrightarrow (dvs\ \mathcal{E}[y])} \quad \begin{array}{l} dvs(x) = C\,x\,{=}rv \\ x \notin \mathsf{HB}(\mathcal{E}), y \notin \mathsf{HB}(\mathcal{E}) \\ \mathsf{fields}(C) = C_1\,f_1 \ldots C_n\,f_n \text{ and } f = f_i \\ \mathsf{get}(rv, i) = y \text{ and } y \in \mathsf{FV}(rv) \end{array}$$

$$\text{(METH-CALL)}\quad \frac{}{(dvs\ \mathcal{E}[x.m(x_1, \ldots, x_n)]) \longrightarrow (dvs\ \mathcal{E}[e[x/\texttt{this}][x_1/y_1] \ldots [x_n/y_n]])} \quad \begin{array}{l} x \notin \mathsf{HB}(\mathcal{E}) \\ x_i \notin \mathsf{HB}(\mathcal{E})\ \forall\ i \in 1..n \\ dvs(x) = C\,x\,{=}rv \\ \mathsf{mbody}(C, m) = \langle y_1 \ldots y_n, e \rangle \end{array}$$

$$\text{(FIELD-ASSIGN)}\quad \frac{}{(dvs\ \mathcal{E}[x.f = y]) \longrightarrow (dvs[x = rv']\ \mathcal{E}[y])} \quad \begin{array}{l} dvs(x) = C\,x\,{=}rv \\ x \notin \mathsf{HB}(\mathcal{E}), y \notin \mathsf{HB}(\mathcal{E}) \\ \mathsf{fields}(C) = C_1\,f_1 \ldots C_n\,f_n \text{ and } f = f_i \\ \mathsf{set}(rv, i, y) = rv' \end{array}$$

$$\text{(DEC)}\quad \frac{(dvs\ e) \longrightarrow (dvs'\ e')}{(dvs\ C\,x\,{=}e\ ds\ e'') \longrightarrow (dvs'\ C\,x\,{=}e'\ ds\ e'')}$$

$$\text{(ALIAS)}\quad \frac{}{(dvs\ C\,x\,{=}y\ ds\ e) \longrightarrow (dvs\ ds\ e)[y/x]}$$

Fig. 3. Reduction rules

name of a field of C, say, the i-th, then the field access is reduced to the reference y stored in this field. The function get returning the i-th field of a right value is defined below (The auxiliary function auxGet also returns the root of a value.)

- $\mathsf{get}(\texttt{new}\ C(x_1, \ldots, x_n), i) = x_i$
- $\mathsf{get}((dvs\ v), i) = y$ if $\mathsf{auxGet}((dvs\ v), i) = \langle x, y \rangle$
- $\mathsf{auxGet}(x, i) = \langle x, \bot \rangle$
- $\mathsf{auxGet}((dvs\ v), i) = \begin{cases} \langle x, y \rangle & \text{if } \mathsf{auxGet}(v, i) = \langle x, \bot \rangle, \mathsf{get}(dvs(x), i) = y \\ \mathsf{auxGet}(v, i) & \text{otherwise} \end{cases}$

The side condition $y \notin \mathsf{HB}(\mathcal{E})$ ensures that there are no inner declarations for y (otherwise y would be erroneously bound), and can be always obtained by α-renaming. For instance, assuming a class table where class `A` has an `int f` field, and class `B` has an `A f` field, the term

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

is reduced to

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

The side condition $y \in \mathsf{FV}(rv)$, requiring that the reference y is not locally declared in rv, prevents scope extrusion, and can always be guaranteed by congruence, that is, by applying rule (CONGR). For instance, without this side condition, the term

```
D x= ( C y= new C() D z= new D(y) z)  x.f
```

would reduce to

```
D x= ( C y= new C() D z= new D(y) z)  y
```

where the last occurrence of `y` would be unbound. Instead, we can take the equivalent term

```
C y= new C() D x= ( D z=new D(y) z)  x.f
```

which correctly reduces to

```
C y= new C() D x= ( D z=new D(y) z)  y
```

Field assignment

In rule (FIELD-ASSIGN), given a field assignment of shape $x.f = y$, the first enclosing declaration for x is found (side condition $x \notin \mathsf{HB}(\mathcal{E})$ ensures that it is the first), and fields of the class C of x are retrieved from the class table. If f is actually the name of a field of C, say, the i-th, then the i-th field of the right value of x is updated to y. We write $dvs[x = rv']$ for the sequence of evaluated declarations obtained from dvs by replacing the right-hand side of the declaration of x by rv' (the obvious formal definition is omitted).

The function set returning a right value where a field has been updated is defined below (the auxiliary function auxSet also returns the root of a value).

- $\mathsf{set}(\texttt{new}\ C(x_1, \ldots, x_n), i, y) = \texttt{new}\ C(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_n)$
- $\mathsf{set}(\texttt{(}dvs\ v\texttt{)}, i, y) = dvs'$ if $\mathsf{auxSet}(\texttt{(}dvs\ v\texttt{)}, i, y) = \langle x, dvs' \rangle$
- $\mathsf{auxSet}(x, i, y) = \langle x, \bot \rangle$
- $\mathsf{auxSet}(\texttt{(}dvs\ v\texttt{)}, i, y) = \begin{cases} \langle x, \texttt{(}dvs[x = \mathsf{set}(rv, i, y)]\ v\texttt{)} \rangle & \text{if } \mathsf{auxSet}(v, i, y) = \langle x, \bot \rangle, \\ & dvs(x) = rv \\ \mathsf{auxSet}(v, i, y) & \text{otherwise} \end{cases}$

The side condition $y \notin \mathsf{HB}(\mathcal{E})$, requiring that there are no inner declarations for the reference y, prevents scope extrusion, and can be always guaranteed by congruence, that is, by applying rule (CONGR). For instance, without this side condition, the term

```
D x=new D(...)  (C y=new C()  x.f=y)
```

would reduce to

```
D x=new D(y)  (C y=new C()  y)
```

Other rules

Rule (CTX) is the usual contextual closure. Rule (CONGR) states that congruence is preserved by reduction, and can be used, as shown above, to reduce a term which otherwise would be stuck, as it happens for α-rule in lambda calculus.

In rule (METH-CALL), given a method invocation of shape $x.m(x_1, \ldots x_n)$, the first enclosing declaration for x is found (side condition $x \notin \mathsf{HB}(\mathcal{E})$ ensures that it is the first), and method m of the class C of x is retrieved from the class table, if actually provided. In this case, the call is reduced to the method body where `this` has been replaced by (the reference to) the receiver object, and parameters have been replaced by arguments. The side condition $x_i \notin \mathsf{HB}(\mathcal{E})\ \forall i = 1..n$ ensures that there are no inner declarations for some argument (which, otherwise, would be erroneously bound), and can be always obtained by α-renaming.

Rule (DEC) avoids to duplicate the above rules for field access, method invocation and field assignment, to handle the case where they occur in the right-hand side of

$$
\begin{array}{lll}
T ::= \mu\ C & & \text{type} \\
\mu ::= \texttt{capsule} \mid \texttt{readable} \mid \epsilon & & \text{type modifier} \\
\Gamma ::= x_1{:}T_1, \ldots, x_n{:}T_n & & \text{type context}
\end{array}
$$

Fig. 4. Types and type contexts

a declaration, rather than in the body, of the block containing that of the receiver object.

In rule (ALIAS), a reference x which is initialized as an alias of another reference y is eliminated by replacing all its occurrences.

3 Type system

Types and type contexts are given in Figure 4.

A type consists in a class name possibly decorated by a *type modifier* which can be either `capsule` or `readable`. A `capsule` expression is expected to reduce to a capsule object, that is, an object with no references from/to the outside (formally, a closed block value), whereas a `readable` expression denotes an object which cannot be modified or aliased. [9]

Type contexts are assumed to be sets (that is, order and repetitions are immaterial), and we use $\emptyset$ for the empty set. Moreover, as usual, they are partial functions from variables to types (that is, no variable occurs more than once). Finally, we assume that such types are either class names, obtained from type annotations in source code, see rule (T-BLOCK), or `readable` types, obtained by weakening current types by rule (T-CAPSULE). That is, whereas `capsule` types can be assigned to expressions, they are not allowed as types of variables. [10]

The typing judgment has shape $\Gamma \vdash e : T$, meaning that expression e has type T in the type context Γ.

The subtyping relation is the reflexive and transitive relation on types induced by

$$\texttt{capsule}\ C \leq\ \ C \leq \texttt{readable}\ C$$

The class table provides type information about methods, abstractly modeled by the following function:

$\mathsf{mtype}(C, m)$ gives, for each method m declared in class C, the triple $\langle T, \mu_0, \mu_1\ C_1 \ldots \mu_n\ C_n \rangle$ consisting of its return type, type modifier for `this`, and parameter types. Type modifiers $\mu_0, \ldots, \mu_n$ are either `readable` or ϵ.

Of course, we assume a well-typed class table, that is, method bodies are expected to be well-typed w.r.t. the corresponding method type. Formally, if $\mathsf{mtype}(C, m) = \langle T, \mu_0, T_1 \ldots T_n \rangle$, then it should be

[9] More precisely, can be temporarily aliased, e.g., when passed as parameter of a method, but cannot be stored within other objects. That is, aliasing here means *static aliasing* in the sense of [13]. As discussed there, static alias can cause unpleasant surprises at an arbitrarily distant point in an execution, whereas dynamic alias has no effects beyond the scope in which it occurs.

[10] The reason is that, to preserve the `capsule` property, a `capsule` variable should be used only once. In this paper for simplicity we prefer to omit this special semantics.

$$\text{(T-CAPSULE)}\ \frac{\mathsf{toReadable}(\Gamma) \vdash e : C}{\Gamma \vdash e : \texttt{capsule}\ C} \qquad \text{(T-SUB)}\ \frac{\Gamma \vdash e : T}{\Gamma \vdash e : T'}\ T \leq T'$$

$$\text{(T-VAR)}\ \frac{}{\Gamma \vdash x : T}\ \Gamma(x) = T \qquad \text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash e : \mu\ C}{\Gamma \vdash e.f : \mu\ C_i}\ \begin{array}{l}\mathsf{fields}(C) = C_1\, f_1 \ldots C_n\, f_n \\ f = f_i\end{array}$$

$$\text{(T-METH-CALL)}\ \frac{\Gamma \vdash e_i : T_i \quad \forall i \in 0..n}{\Gamma \vdash e_0.m(e_1, \ldots, e_n) : T}\ \begin{array}{l}T_0 = \mu\ C \\ \mathsf{mtype}(C, m) = \langle T, \mu, T_1 \ldots T_n \rangle\end{array}$$

$$\text{(T-FIELD-ASSIGN)}\ \frac{\Gamma \vdash e : C \quad \Gamma \vdash e' : C'}{\Gamma \vdash e.f = e' : C'}\ \begin{array}{l}\mathsf{fields}(C) = C_1\, f_1 \ldots C_n\, f_n \\ f = f_i, C' = C_i\end{array}$$

$$\text{(T-NEW)}\ \frac{\Gamma \vdash e_i : C_i \quad \forall i \in 1..n}{\Gamma \vdash \texttt{new}\ C(e_1, \ldots, e_n) : C}\ \mathsf{fields}(C) = C_1\, f_1 \ldots C_n\, f_n$$

$$\text{(T-BLOCK)}\ \frac{\Gamma[\Gamma'] \vdash e_i : C_i \quad \forall i \in 1..n \quad \Gamma[\Gamma'] \vdash e : T}{\Gamma \vdash \texttt{(}\, C_1\, x_1\, \texttt{=}e_1 \ldots C_n\, x_n\, \texttt{=}e_n\ e\texttt{)} : T}\ \Gamma' = x_1{:}C_1 \ldots x_n{:}C_n$$

Fig. 5. Typing rules

$\mathsf{mbody}(C, m) = \langle x_1 \ldots x_n, e \rangle$, and $\Gamma \vdash e : T$ with $\Gamma = \texttt{this}{:}\mu_0\ C, x_1{:}T_1, \ldots, x_n{:}T_n$.

Typing rules are given in Figure 5.

Rule (T-CAPSULE) states that an expression can be typed as `capsule` if all its free variables are used as readable. We write $\mathsf{toReadable}(\Gamma)$ for the type context obtained from Γ by setting all type modifiers to `readable`. Rule (T-SUB) is the usual subsumption. Other rules are mostly standard, apart that they model the expected behaviour of type modifiers. Notably, in rule (T-FIELD-ACCESS), the type modifier is propagated to fields. For instance, fields of a readable object are readable as well. In rule (T-FIELD-ASSIGN), neither the receiver nor the right-hand-side expression can be readable. In rule (T-NEW), analogously, values assigned to fields cannot be readable, since saving a reference as field of an object introduces an alias.

In rule (T-BLOCK), we write $\Gamma[\Gamma']$ for the concatenation of Γ and Γ' where, for the variables occurring in both domains, Γ' takes precedence.

It should be clear how to extend the formal definition to handle primitive types, used in previous and following examples. Briefly, modifiers make no sense on such types, which are simply used in the standard way. For instance, in the premise of rule (T-NEW) the types of constructor arguments could be primitive types as well, whereas in rule (T-METH-CALL) the type of method receiver could not.

We illustrate now how the rule (T-CAPSULE) works. Consider the following term:

```
D z= new D(0)
C x= ( D y= new D(z.f+1)   new C(y,y) )
x
```

The inner block (right-hand side of the declaration of `x`) can be typed `capsule`, since free variable `z` is only used as readable (neither modified nor aliased). Formally, we can apply rule (T-CAPSULE). Indeed, the block reduces to

`( D y= new D(1) C x = new C(y,y) x )` which is a capsule.

As a counterexample, consider the following term:

```
D z= new D(0)
C x= ( D y= z   new C(y,y) )
x
```

Here the inner block cannot be typed `capsule`, since `z` is internally aliased. Formally, we cannot apply (T-CAPSULE) on the block, since we should typecheck the block with $\Gamma = \texttt{z:readable D}$, while (T-BLOCK) requires `D` as type of `z`. Indeed, the block reduces to `(new C(z,z))` which is not a capsule.

4 Results

We use the abbreviations $e \longrightarrow$ for $e \longrightarrow e'$ for some e', $\vdash e : T$ for $\emptyset \vdash e : T$, and $\vdash e$ for $\vdash e : T$ for some T.

The soundness theorem states that reduction of well-typed closed expressions does not get stuck.

Theorem 4.1 (Soundess) *If $\vdash e$, and $e \longrightarrow^\star e'$, then either e' is a value, or $e' \longrightarrow$.*

Soundness is obtained, as usual, as a consequence of progress and subject reduction theorems. Note that, since our operational model is a pure calculus, in the proofs we do not need invariants on auxiliary structures such as memory.

Theorem 4.2 (Progress) *If $\vdash e$, then either e is a value, or $e \longrightarrow$.*

The progress theorem is obtained as an immediate corollary of extended progress.

Theorem 4.3 (Extended Progress) *If $\Gamma \vdash e : T$, then one of the following cases holds:*

(i) *e is a value, with $\mathsf{FV}(e) \subseteq \mathsf{dom}(\Gamma)$*

(ii) *$e \longrightarrow$*

(iii) *$e = \mathcal{E}[x.f]$, $x \notin \mathsf{HB}(\mathcal{E})$, and $x \in \mathsf{dom}(\Gamma)$*

(iv) *$e = \mathcal{E}[x.m(xs)]$, $x \notin \mathsf{HB}(\mathcal{E})$, and $x \in \mathsf{dom}(\Gamma)$*

(v) *$e = \mathcal{E}[x.f = y]$, $x \notin \mathsf{HB}(\mathcal{E})$, and $x \in \mathsf{dom}(\Gamma)$.*

Theorem 4.4 (Subject reduction) *If $\Gamma \vdash e : T$, and $e \longrightarrow e'$, then $\Gamma \vdash e' : T$.*

In addition to soundness, we state that the `capsule` modifier actually ensures the expected behaviour. A nice consequence of our non standard operational model is that this can be easily formally expressed and proved, as shown below.

Informally, a capsule is a reachable object subgraph where nodes cannot be reached from the outside. In our model, where reachable object subgraphs are directly represented by language values, a capsule is simply a closed value. Hence the fact that an expression of `capsule` type actually reduces to a capsule can be stated as in Theorem 4.6 below.

Let $\mathsf{typectx}(\mathcal{E})$ be the type context extracted from a context $\mathcal{E}$, whose trivial definition is given in the Appendix. Moreover, to trace the reduction of an expression

inside a context, let us assume that in the result $\mathcal{E}[e]$ of filling the hole of a context, we can still recover the subterm e (for instance, we can replace the hole by $[e]$, with square brackets immaterial for reduction rules).

Lemma 4.5 *If* $\vdash \mathcal{E}[e]$, $\mathsf{typectx}(\mathcal{E}) \vdash e : T$, *and* $\mathcal{E}[e] \longrightarrow \mathcal{E}'[e']$, *then* $\vdash \mathcal{E}'[e']$ *and* $\mathsf{typectx}(\mathcal{E}') \vdash e' : T$.

Theorem 4.6 (Capsule) *If* $\vdash \mathcal{E}[e]$, $\mathsf{typectx}(\mathcal{E}) \vdash e :$ `capsule` C, *and* $\mathcal{E}[e] \longrightarrow^\star \mathcal{E}'[v]$, *then* v *is closed.*

Proof. We know that $\mathsf{typectx}(\mathcal{E}') \vdash v :$ `capsule` C by Lemma 4.5. Set $\Gamma' = \mathsf{typectx}(\mathcal{E}')$. By structural induction on v.

x Empty case. Indeed, we can assign a `capsule` type to a variable only by rule (T-CAPSULE) or (T-VAR). However, to apply rule (T-CAPSULE) we should derive $\mathsf{toReadable}(\Gamma') \vdash x : C$, which is not possible, and to apply rule (T-VAR) we should have $\Gamma'(x) =$ `capsule` C, whereas type contexts do not assign `capsule` types.

(*dvs v*) We can assign a `capsule` type to a block only by rule (T-CAPSULE) or (T-BLOCK). If we have applied rule (T-CAPSULE), then all free variables are required to be readable. Free variables in block values only occur as values of fields (the root variable is necessarily bound), which cannot be readable, hence the block has no free variables. If we have applied rule (T-BLOCK), then v has a `capsule` type as well, hence by inductive hypothesis is closed. Hence, (*dvs v*) is equivalent to v by congruence rule (GARBAGE).

□

5 Conclusion

We have presented an imperative calculus where the block construct, introducing local variable declarations, also plays the role of store when such declarations have been evaluated. In this way we are able to define a pure semantics with no auxiliary data structure, where aliasing properties can be directly expressed at the syntax level, allowing much simpler reasoning.

To illustrate this advantage, let us consider typing rule (T-CAPSULE) in Figure 5. Here we want to express that an expression e, subterm of a program, can be typed `capsule` if it can modify only its local objects. Objects which are reachable from other parts of the program, instead, can only be used as `readable`. In our model, the objects reachable from other parts of the program are simply those denoted by the free variables in e, whose type is required indeed to be `readable` in the premise of the rule, whereas the local objects are those denoted by local variables declared in e. In other terms, the portion of memory only reachable from e is encoded in e itself. In a conventional model with global memory, to express the same property, we should, first of all, type the memory locations as well, and add invariants on the memory to prove subject reduction. Then, we should require to use only as `readable` the locations which are reachable from other parts of the program. However, the information that some locations are only reachable from e is lost in the global memory. To be concrete, consider the following example:

```
A a= new A(...)  B b=( C c=new C()   c.foo() )
```

In the conventional model, this program is reduced by first adding to the memory two new locations, say ι_a and ι_c, which are then used to replace variables a and c, respectively. We then get to execute ι_c`.foo()`. To type this expression, we would use the following judgement: $\emptyset; \iota_a{:}A, \iota_c{:}C \vdash \iota_c$`.foo()`$:B$. As you can see, there is no information about how ι_c is used inside the rest of the program. For example ι_c may be in the reachable graph of ι_a. In our approach `C c=new C()` is kept in place, and we use the following judgment: $a{:}A \vdash$ `( C c=new C() c.foo() )`$:B$. Here the information that `c` is used only in the block is implicit from the block scope. Our aim here is to formalize the execution model we have in mind in a simple and abstract way. To this end, we define a congruence on terms which can be used to reduce a term which otherwise would be stuck. Of course an implementation should detect when and how to apply a congruence rule, very much in the same way an implementation of the lambda calculus must apply α-conversion when necessary.

The fact that aliasing properties can be expressed at the syntax level should more easily allow the implementation of an interpreter for the calculus into a theorem prover. Indeed, e.g., the Key theorem prover [1] uses an approach, called *abstract object creation*, where parallel update (a kind of runtime expression) is added to the language and used to represent the store inside the code: all the object creations and field updates are preserved and consulted by field accesses.

The Racket stepper [7] is a program execution visualization tool that simulates a pure calculus out of a language with mutable bindings defined in the conventional way, relying on specially forged runtime expressions. The stepper accurately models the functional setting of Racket. In the stepper, bindings are lifted to the top level; while it does not currently step through mutable bindings, the authors argue that it can be easily extended in such a way. A more recent development, PLT Redex [9], could be applied to a pure calculus like ours.

The fact that language values are blocks with mutually recursive declarations is reminiscent of cyclic lambda calculi, see, e.g., [4,3]. Indeed, in both cases a declared variable is not replaced by its value, as it happens with standard `let`, but the association is kept and used when necessary. However, in cyclic lambda calculi there is a different aim (mainly to provide an efficient reduction strategy), and, on the technical side, there is no equivalent of the problem that reading/assigning a field can cause scope extrusion.

Felleisen's syntactic theory of state [10,11] mangles the store in the expressions, but it relies on labelled values, a kind of runtime expression modelling the value and the location address at the same time.

In future work, we plan to use (variants of) the calculus presented in this paper as a basis to express and formally verify different properties of object graphs, among those proposed in the wide literature about ownership, see, e.g., [6]. A more ambitious goal will be to investigate (a form of) Hoare logic on top of this model. We believe that the hierarchical structure of our memory representation should help local reasoning, allowing specifications and proofs to mention only the relevant portion, analogously to what is achieved by separation logic [16].

We also plan to formally state and prove the equivalence of the calculus with conventional imperative models.

Acknowledgements

We warmly thank the anonymous referees for their very useful comments. In particular, one referee pointed out the analogy with scope extrusion in the pi-calculus, leading to a better formulation of congruence and reduction rules. We also thank Lindsay Groves, co-author of a preliminary version of this work [17], with focus on didactic applications.

References

[1] Wolfgang Ahrendt, Frank S. de Boer, and Immo Grabe. Abstract object creation in dynamic logic. In *FM 2009: Formal Methods*, pages 612–627, 2009.

[2] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, 1997.

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

[4] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journ. of Functional Programming*, 7(3):265–301, 1997.

[5] David Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP'03 - Object-Oriented Programming*, volume 2473 of *Lecture Notes in Computer Science*, pages 176–200. Springer, 2003.

[6] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1998*, pages 48–64, 1998.

[7] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *ESOP'01 - European Symposium on Programming*, pages 320–334, 2001.

[8] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In Erik Ernst, editor, *ECOOP'07 - Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer, 2007.

[9] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[10] Matthias Felleisen and Daniel P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3):243–287, 1989.

[11] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[12] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In Gary T. Leavens and Matthew B. Dwyer, editors, *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2012)*, pages 21–40. ACM Press, 2012.

[13] John Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991*, pages 271–285. ACM Press, 1991.

[14] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[15] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[16] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. IEEE Symp. on Logic in Computer Science 2002*, pages 55–74. IEEE Computer Society, 2002.

[17] Marco Servetto and Lindsay Groves. True small-step reduction for imperative object-oriented languages. *FTfJP'13- Formal Techniques for Java-like Programs*, 2013.

[18] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix - safe modular circular initialisation with placeholders and placeholder types. In Giuseppe Castagna, editor, *ECOOP'13 - Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 205–229. Springer, 2013.

[19] Marco Servetto, David J. Pearce, Lindsay Groves, and Alex Potanin. Balloon types for safe parallelisation over arbitrary object graphs. In *WODET 2014 - Workshop on Determinism and Correctness in Parallel Programming*, 2013.

A Auxiliary definitions

$\mathsf{HB}(\mathcal{E})$:

$\mathsf{HB}([\,]) = \emptyset$
$\mathsf{HB}(\mathcal{E}.f) = \mathsf{HB}(\mathcal{E}.m(es)) = \mathsf{HB}(x.m(xs, \mathcal{E}, es)) = \mathsf{HB}(\mathcal{E}.f = e') = \mathsf{HB}(x.f = \mathcal{E}) = \mathsf{HB}(\texttt{new}\ C(xs, \mathcal{E}, es)) = \mathsf{HB}(\mathcal{E})$
$\mathsf{HB}(\texttt{(}dvs\ C\ x\ \texttt{=}\mathcal{E}\ ds\ e\texttt{)}) = \mathsf{HB}(\mathcal{E}) \cup \mathsf{dom}(dvs) \cup \mathsf{dom}(ds)$
$\mathsf{HB}(\texttt{(}dvs\ \mathcal{E}\texttt{)}) = \mathsf{HB}(\mathcal{E}) \cup \mathsf{dom}(dvs)$

$\mathsf{typectx}(\mathcal{E})$:

$\mathsf{typectx}([\,]) = \emptyset$
$\mathsf{typectx}(\mathcal{E}.f) = \mathsf{typectx}(\mathcal{E}.m(es)) = \mathsf{typectx}(x.m(xs, \mathcal{E}, es)) = \mathsf{typectx}(\mathcal{E}.f = e') = \mathsf{typectx}(x.f = \mathcal{E}) = \mathsf{typectx}(\texttt{new}\ C(xs, \mathcal{E}, es)) = \mathsf{typectx}(\mathcal{E})$
$\mathsf{typectx}(\texttt{(}dvs\ C\ x\ \texttt{=}\mathcal{E}\ ds\ e\texttt{)}) = \mathsf{typectx}(dvs\ ds)[\mathsf{typectx}(\mathcal{E})]$
$\mathsf{typectx}(\texttt{(}dvs\ \mathcal{E}\texttt{)}) = \mathsf{typectx}(dvs)[\mathsf{typectx}(\mathcal{E})]$
$\mathsf{typectx}(C_1\ x_1\ \texttt{=}e_1 \ldots C_n\ x_n\ \texttt{=}e_n) = x_1{:}C_1 \ldots x_n{:}C_n$

$\mathsf{FV}(e)$:

$\mathsf{FV}(x) = \{x\}$
$\mathsf{FV}(e.f) = \mathsf{FV}(e)$
$\mathsf{FV}(e_0.m(e_1, \ldots, e_n)) = \mathsf{FV}(e_0) \cup \ldots \cup \mathsf{FV}(e_n)$
$\mathsf{FV}(e.f = e') = \mathsf{FV}(e) \cup \mathsf{FV}(e')$
$\mathsf{FV}(\texttt{new}\ C(e_1, \ldots, e_n)) = \mathsf{FV}(e_1) \cup \ldots \cup \mathsf{FV}(e_n)$
$\mathsf{FV}(\texttt{(}ds\ e\texttt{)}) = (\mathsf{FV}(ds) \cup \mathsf{FV}(e)) \setminus \mathsf{dom}(ds)$
$\mathsf{FV}(C_1\ x_1\ \texttt{=}e_1 \ldots C_n\ x_n\ \texttt{=}e_n) = \mathsf{FV}(e_1) \cup \ldots \cup \mathsf{FV}(e_n)$

$e[y/x]$:

$x[y/x] = y$
$z[y/x] = z$ if $z \neq x$
$e.f[y/x] = e[y/x].f$
$e_0.m(e_1, \ldots, e_n)[y/x] = e_0[y/x].m(e_1[y/x], \ldots, e_n[y/x])$
$(e.f = e')[y/x] = e[y/x].f = e'[y/x]$
$\texttt{new}\ C(e_1, \ldots, e_n)[y/x] = \texttt{new}\ C(e_1[y/x], \ldots, e_n[y/x])$
$\texttt{(}ds\ e\texttt{)}[y/x] = \texttt{(}ds[y/x]\ e[y/x]\texttt{)}$ if $x \notin \mathsf{dom}(ds), y \notin \mathsf{dom}(ds)$
$\texttt{(}ds\ e\texttt{)}[y/x] = \texttt{(}ds\ e\texttt{)}$ if $x \in \mathsf{dom}(ds)$
$(C_1\ x_1\ \texttt{=}e_1 \ldots C_n\ x_n\ \texttt{=}e_n)[y/x] = C_1\ x_1\ \texttt{=}e_1[y/x] \ldots C_n\ x_n\ \texttt{=}e_n[y/x]$