Types for Immutability and Aliasing Control

Paola Giannini[1], Marco Servetto[2], and Elena Zucca[3]

[1] Università del Piemonte Orientale, Italy
giannini@di.unipmn.it
[2] Victoria University of Wellington, New Zealand
marco.servetto@ecs.vuw.ac.nz
[3] Università di Genova, Italy
elena.zucca@unige.it

1 Introduction

In mainstream languages with state and explicit mutations, unwanted aliasing relations are common bugs. This is exasperated by concurrency mechanisms, since unpredicted aliasing can induce unplanned/unsafe communication points between threads.

For these reasons, a massive amount of research, see, e.g., [16,9,11,6], has been devoted to make programming with side-effects easier to maintain and understand, notably using *type modifiers* to control state access. In this paper, we will use five type modifiers (`mut`, `imm`, `capsule`, `lent`, `read`) to obtain a fine-tuned control of immutability and aliasing properties.

Let us consider the store as a graph, where nodes contain records of fields, that may be references to other nodes. Each node determines a subgraph, that of all the nodes reachable from it. Let x be a reference to a node in the graph. Is x is *mutable*, then it can be freely used, and we cannot make any assumption on it. If x is *immutable*, then $x.f{=}e$ is not allowed, and we can assume that the subgraph reachable from x will not be modified through any other reference. An immutable reference can be safely shared also in a multithreaded environment. If x is a *capsule*, then we can assume that the subgraph reachable from x is an isolated portion of store, that is, all its (non immutable) nodes cannot be reached through other references. Capsule references can be used only once, as both mutable or immutable, e.g., to initialize a mutable/immutable reference. In other words, a capsule reference can be seen as a reference whose destiny has not been decided yet. Moreover, if a capsule reference x is assigned to a mutable one y, then (in that moment) no part of this subgraph can be updated through a reference different from y. This allows to identify mutable state that can be safely handled by a thread. If x is *lent*, then it can be used in a restricted way, so that

V. Biló, A. Caruso (Eds.): ICTCS 2016, Proceedings of the 17th Italian Conference on Theoretical Computer Science, 73100 Lecce, Italy, September 7–9 2016, pp. 62–74 published in CEUR Workshop Proceedins Vol-1720 at http://ceur-ws.org/Vol-1720

no aliasing can be introduced that would make possible to reach the subgraph of x. Finally, *readable* references can neither be modified nor aliased. These last two modifiers ensure intermediate properties used to derive the immutable and capsule properties.

Whereas (variants of) such modifiers have appeared in previous literature (see Sect.4 for a discussion on related work), there are two key novelties w.r.t. similar proposals. First, the expressivity of the type system is greatly enhanced. Indeed, modifiers restrict, as described above, the use of a reference in a context, but this restriction can be escaped by *promotion*, at the price of restricting the use of other references. Second, it is possible to express and check aliasing and immutability property directly on source terms, without introducing invariants on an auxiliary structure which mimics physical memory. Indeed, we adopt an innovative execution model [12,4,14] for imperative languages which, differently from traditional ones, is a *pure calculus*.

In this extended abstract, due to space constraints, we focus on the type system, and only illustrate the calculus by simple examples. We refer to the full paper [8] for reduction rules, more examples and proofs of results.

The rest of the paper is organized as follows: we provide syntax and an informal introduction in Sect.2, formalize the type system and state results in Sect.3, discuss related work, paper contribution and further work in Sect.4.

2 Informal introduction

Syntax and types are given in Fig.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 xs is defined by $xs ::= \epsilon \mid x\ xs$, where ϵ denotes the empty sequence.

cd	::= `class` C `{`$fds\ mds$`}`	class declaration
fd	::= `imm` $C\ f$ | `mut` $C\ f$ | `int`	field declaration
md	::= $T\ m\ \mu$ `(`$T_1\ x_1, \ldots, T_n\ x_n$`)` `{return` e`}`	method declaration
e	::= x | $e.f$ | $e.m$`(`es`)` | $e.f$`=`e | `new` C`(`es`)` | `{`$ds\ e$`}`	expression
d	::= $T\,x$`=`e;	variable declaration
T	::= $\mu\ C$ | `int`	type
μ	::= `imm` | `mut` | `capsule` | `lent` | `read`	type modifier
dv	::= $T\,x$`=`v;	evaluated declaration
u,v	::= x | `new` C`(`xs`)` | `{`$dvs\ x$`}` | `{`dvs `new` C`(`xs`)}`	value

Fig. 1: Syntax and types

The syntax mostly follows Java and Featherweight Java (FJ) [10]. A class declaration consists of a class name, a sequence of field declarations and a sequence

of method declarations. A field declaration consists of a field type and a field name. A method declaration consists, as in FJ, of a return type, a method name, a list of parameter names with their types, and a body which is an expression. However, there is an additional component: the type modifier for `this`, which is placed after the method name. As in FJ, we assume for each class a canonical constructor whose parameter list exactly corresponds to the class fields, and we assume no multiple declarations of classes in a class table, fields and methods in a class declaration.

For expressions, in addition to the standard constructs of imperative OO languages, we have *blocks*, which are sequences of variable declarations, followed by a *body* which is an expression. Variable declarations consist of a type, a variable and an initialization expression. Types are class names decorated by a *type modifier*. We also include `int` as an example of primitive type, but we do not formally model related operators used in the examples, such as integer constants and sum. We assume no multiple declarations for variables in a block.

Values are references x, *object states*, that is, constructor invocations where arguments are references, or blocks in which declarations are evaluated, that is, initialization expressions are values, and the body is a reference or an object state.

Blocks are a fundamental construct of our language, since sequences of local variable declarations, when evaluated, are used to directly represent store in the language itself. For instance[4], assuming that class `B` has a `mut` field of type `B`:

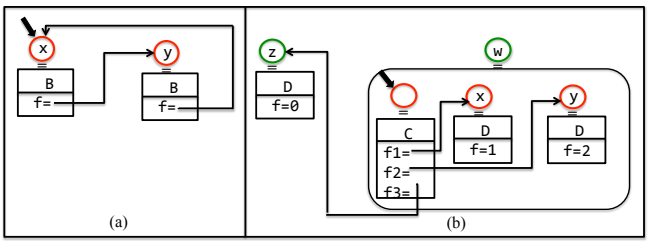

Fig. 2: Graphical representation of the store

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

the two declarations can be seen as a store where `x` denotes an object of class `B` whose field is `y`, and conversely, as shown in Fig.2(a). The whole block denotes a store with an entry point (graphically represented by a thick arrow), that is, an object.

Moreover, store is *hierarchical*, rather than flat as it usually happens in models of imperative languages. For instance, assuming that class `C` has two `mut` and one `imm D` fields, and class `D` has an integer field, the following is a store:

```
imm D z= new D(0);
imm C w= {mut D x= new D(1); mut D y= new D(2); new C(x,y,z)}
```

Here, the value associated to `w` is a block introducing local declarations, that is, in turn a store, as shown in Fig.2(b). The advantage of this hierarchical shape is that it models in a simple and natural way constraints about aliasing among objects, notably:

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

– the fact that an object is not referenced from outside some enclosing object is directly modeled by the block construct: for instance, the objects denoted by `x` and `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[5]: for instance, the object denoted by `w` is not closed, since it refers to the external object `z`.

In the graphical representation, variables circled in red are mutable references, whereas the ones circled in green are immutable. Note that the reference corresponding to `new C(x,y,z)` is anonymous. Note also that, in this example, mutable variables in the local store of `w` are not visible from the outside. This models in a natural way the fact that the portion of store denoted by `w` is indeed immutable, as will be detailed in the sequel.

We illustrate now the meaning of the modifiers `mut`, `imm`, and `capsule`. A mutable variable refers to a portion of store that can be modified during execution. For instance, the block

```
mut B x= new B(y);   mut B y= new B(x);   x.f = x
```

reduces to

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

We give a graphical representation of this reduction in Fig.3. In the graphical representation, we highlight in grey expressions which are not values. So in (a) the body of the block is the expression `x.f=x`, whose evaluation modifies the field `f` of `x`, and returns `x`. The result of the reduction is shown in (b).

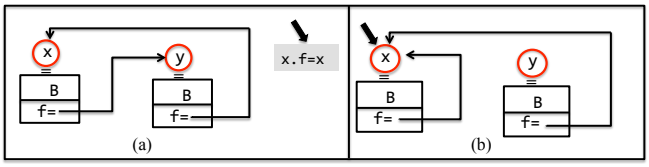

Fig. 3: Example of reduction (1)

Variables declared immutable, instead, once they have an associated value, cannot be modified. Immutability is *deep*, that is, all the nodes of the reachable object graph of an immutable reference are immutable themselves. Therefore, in the enclosing scope of the declaration

```
imm C w= {mut D x= new D(1); mut D y= new D(2); new C(x,y,z)}
```

the variable `z` must be declared `imm`, and we cannot have an assignment to a field of `w`.

A variable declared `capsule` refers to an *isolated* portion of store, where local objects can freely reference each other but for which the current variable is the only external reference. For instance:

```
capsule B z = { mut B x= new B(y);   mut B y= new B(x);   x }
```

[5] In other words, our calculus smoothly integrates memory representation with shadowing and α-conversion.

The internal objects denoted by `x` and `y` can be only be accessed through `z`. A capsule variable is a temporary reference, to be used once and for all to "move" an isolated portion of store to another node in the store. To get more flexibility, external immutable references are freely allowed. For instance, in the example above of the declaration of `w`, the inizialization expression has a `capsule` type. In our type system, capsule types are subtypes of both mutable and immutable types. Hence, capsule expressions can initialize both mutable and immutable references. However, to preserve the capsule property, we need a *linearity constraint*: in well-formed expressions capsule references can occur at most once in their scope.

Consider the term

```
mut D y=new D(0);
capsule C z={mut D x=new D(y.f=y.f+1); new C(x,x)}
```

In Fig.4(a) we have a graphical representation of this term, where the variable circled in blue is a capsule reference.

The evaluation of the expression on the right-hand side of `x` starts by evaluating `y.f+1`, which triggers the evaluation of `y.f`. The result is shown in (b), then the sum `0+1` is evaluated, returning `1`, as shown in (c). The evaluation of the field assignment `y.f=1`, updates the field `f` of `y` to `1`, and `1` is returned. Since `new D(1)` is a value, the whole term is fully evaluated, and it is shown in (d).

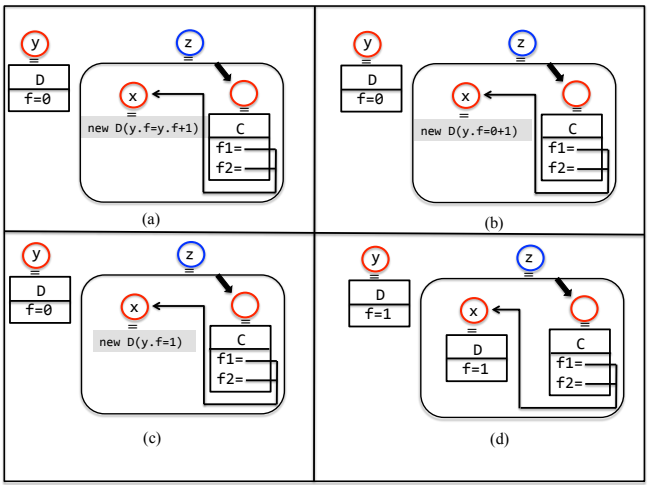

Fig. 4: Example of reduction (2)

To be able to typecheck more expressions as `capsule` or `imm`, we introduce the `lent` and `read` modifiers. References with such modifiers can be used in a restricted way. Notably, they can be used to access fields, but cannot be used either on the left or the right-hand side of an assignment or in an object creation. For `lent` references this restriction is not permanent, in the sense that it is possible to freely use a `lent` reference in a subexpression at the price of restricting other references, as will be detailed in the following section. Clearly, wherever a lent reference is required we could use a mutable one, so there is a subtyping relation between `lent` and `mut` modifiers. In some cases it is possible to move the type of an expression against the subtype hierarchy, that is, to *promote* an expression. Notably, a `mut` expression can be promoted to `capsule`, and a `read` expression can be promoted to `imm`, provided that some of the free variables in

the expression are used in a restricted way. The situation is graphically depicted in Fig.5. Promotions will be described in the next section.

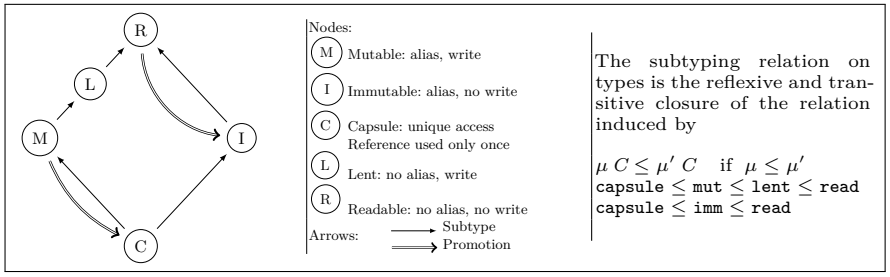

Fig. 5: Type modifiers and their relationships

3 Type system and results

A *type context* $\Delta = \Gamma; xss; ys$ consists of a usual *assignment of types* to variables $\Gamma = x_1{:}T_1, \ldots, x_n{:}T_n$, and two additional components xss and ys. Accordingly with our convention, xss is a sequence $xs_1 \ldots xs_n$ of sequences of variables, and ys is a sequence of variables. All such sequences are assumed to be sets (that is, order and repetitions are immaterial).

Sets $xs_1 \ldots xs_n$ are pairwise disjoint, and their elements are variables of `mut` type in Γ, hence they describe a partition of such variables in $n+1$ sets, called *groups*, the last being the set of those not belonging to any xs_i, called the *current group*. Elements of ys are variables of `mut`, `lent` or `read` type in Γ. The sets xss and ys model restrictions on the variables in Γ, as motivated below.

The type system must assure that, if for an expression e we derive type T, then the evaluation of the expression produces a value with the property expressed by T. For instance, if the following two variable declarations are well-typed `capsule B z1 = e1; imm B z2 = e2;` then the evaluation of the expressions `e1` and `e2` in the context of a well-typed program should produce values which are capsule and immutable, respectively, in the sense introduced in the previous section.

Let us discuss, for instance, when an expression could be safely typed `capsule`. Obviously, this is safe for an expression with no free variables, or where all the free variables are immutable. However, this requirement is too strong. For instance, consider the following sequence of declarations:

```
mut D y=new D(0);capsule C z={mut D x=new D(y.f); new C(x,x)};
```

The inner block (right-hand side of the declaration of `z`) can be typed `capsule`, even though there is a mutable free variable `y`, since this variable is only used in a field access. However, if we had `y` instead of `new D(y.f)`, the declaration would not be well-typed, since through the variable `y` we would refer to the reachable graph of `z`.

Formally, the inner block can be typechecked in a type context where *xss* consists of only one singleton group containing variable `y`. This means that this variable, which was originally `mut`, is restricted to `lent` when typechecking the inner block, hence no aliasing can be introduced between `y` and the final result of the block.

In general, in the *typing judgment* $\Gamma; xss; ys \vdash e : T$, the sets *xss* and *ys* model restrictions on aliasing which could be possibly introduced by the evaluation of the expression. That is, if *xs* and *xs′* are two different groups in *xss*, and if their reachable graphs were disjoint before the evaluation of the expression, then after the evaluation they would be still disjoint. Analogously, no aliasing can be introduced between the reachable graph of any group *xs* in *xss* and that of the result. The same constraint holds for the portion of store reachable from the variables in *ys*. These restrictions are achieved as follows:

- a `mut` variable in Γ which belongs to a group in *xss* is *lent-restricted*, that is, can only be used as `lent`
- a `read`, `lent` or `mut` variable in Γ which belongs to *ys* is *strongly-restricted*, that is, cannot be used at all.

Variables become restricted as an effect of applying *promotion* rules (T-CAPSULE) and (T-IMM), and restricted variables can be temporarily unrestricted by applying (T-SWAP) and (T-UNRST) rules, as will be explained in detail later.

Typing rules are given in Fig.6. In the rules we use information extracted from the class table, which is modelled, as usual, by the following functions:

- $\mathsf{fields}(C)$ gives, for each declared class C, the sequence of its fields declarations
- $\mathsf{method}(C, m)$ gives, for each method m declared in class C, the tuple $\langle T, \mu, T_1\ x_1 \ldots T_n\ x_n, e\rangle$ consisting of its return type, type modifier for `this`, parameters, and body.

We assume method bodies to be well-typed w.r.t. the type annotations in the method declaration. Formally, if
$\mathsf{method}(C, m) = \langle T, \mu, T_1\ x_1 \ldots T_n\ x_n, e\rangle$, then it should be $\Gamma; \emptyset; \emptyset \vdash e : T$, with $\Gamma = \texttt{this}{:}\mu\ C, x_1{:}T_1, \ldots, x_n{:}T_n$.

Rules (T-CAPSULE) and (T-IMM) model *promotions*, that is, can be used to promote an expression from a more general to a more specific type, under the conditions that some free variables in the expression are restricted. There are two kinds of promotion:

- `mut` ⇒ `capsule`
  As shown in rule (T-CAPSULE), an expression can be typed `capsule` in $\Gamma; xss; ys$ if it can be typed `mut` by lent-restricting the current group of mutable variables (*xs*). Formally, this group is added to *xss*.
- `read` ⇒ `imm`
  As shown in rule (T-IMM), an expression can be typed `imm` in $\Gamma; xss; ys$ if it can be typed `read` by strongly-restricting, rather than *ys* only, *all* currently available mutable, lent, and readable variables ($\mathsf{dom}^{\geq \texttt{mut}}(\Gamma)$). The current group of mutable variables (*xs*) is also added as a new group to *xss*.

Note that set difference in the side conditions makes sense since xss is assumed to be a set.

$$\text{(T-CAPSULE)}\ \frac{\Gamma; xss\ xs; ys \vdash e : \texttt{mut}\ C}{\Gamma; xss; ys \vdash e : \texttt{capsule}\ C}\ xs = \mathsf{dom}^{\texttt{mut}}(\Gamma) \backslash xss$$

$$\text{(T-IMM)}\ \frac{\Gamma; xss\ xs; \mathsf{dom}^{\geq \texttt{mut}}(\Gamma) \vdash e : \texttt{read}\ C}{\Gamma; xss; ys \vdash e : \texttt{imm}\ C}\ xs = \mathsf{dom}^{\texttt{mut}}(\Gamma) \backslash xss$$

$$\text{(T-SWAP)}\ \frac{\Gamma; xss\ xs'; ys \vdash e : \mu\ C}{\Gamma; xss\ xs; ys \vdash e : \mu'\ C}\ xs' = \mathsf{dom}^{\texttt{mut}}(\Gamma) \backslash (xss\ xs)\ \ \mu' = \begin{cases} \texttt{lent} & \text{if } \mu = \texttt{mut} \\ \mu & \text{otherwise} \end{cases}$$

$$\text{(T-UNRST)}\ \frac{\Gamma; xss; \emptyset \vdash e : \mu\ C}{\Gamma; xss; ys \vdash e : \mu\ C}\ \mu \leq \texttt{imm} \qquad \text{(T-SUB)}\ \frac{\Delta \vdash e : T}{\Delta \vdash e : T'}\ T \leq T'$$

$$\text{(T-VAR)}\ \frac{}{\Gamma; xss; ys \vdash x : T'}\ \Gamma(x) = T \ \wedge\ x \notin ys\ \ T' = \begin{cases} \texttt{lent}\ C & \text{if } T = \texttt{mut}\ C \wedge x \in xss \\ T & \text{otherwise} \end{cases}$$

$$\text{(T-FIELD-ACCESS)}\ \frac{\Delta \vdash e : \mu\ C}{\Delta \vdash e.f_i : T'_i}\ \mathsf{fields}(C) = T_1\ f_1 \ldots T_n\ f_n\ \ T'_i = \begin{cases} \mu\ C_i & \text{if } T_i = \texttt{mut}\ C_i \\ T_i & \text{otherwise} \end{cases}$$

$$\text{(T-METH-CALL)}\ \frac{\Delta \vdash e_i : T_i \quad \forall i \in 0..n}{\Delta \vdash e_0.m(e_1, \ldots, e_n) : T}\ T_0 = \mu\ C\ \ \mathsf{method}(C, m) = \langle T, \mu, T_1\ x_1 \ldots T_n\ x_n, e \rangle$$

$$\text{(T-FIELD-ASSIGN)}\ \frac{\Delta \vdash e : \texttt{mut}\ C \quad \Delta \vdash e' : T_i}{\Delta \vdash e.f_i = e' : T_i}\ \mathsf{fields}(C) = T_1\ f_1 \ldots T_n\ f_n$$

$$\text{(T-NEW)}\ \frac{\Delta \vdash e_i : T_i \quad \forall i \in 1..n}{\Delta \vdash \texttt{new}\ C(e_1, \ldots, e_n) : \texttt{mut}\ C}\ \mathsf{fields}(C) = T_1\ f_1 \ldots T_n\ f_n$$

$$\text{(T-BLOCK)}\ \frac{\Gamma[\Gamma']; xss'; ys \vdash e_i : T_i \quad \forall i \in 1..n \qquad \Gamma[\Gamma']; xss'; ys \vdash e : T}{\Gamma; xss; ys \vdash \{\, T_1\ x_1 = e_1; \ldots T_n\ x_n = e_n;\ e \,\} : T}\ \begin{array}{l} \Gamma' = x_1 : T_1, \ldots, x_n : T_n \\ xss = xss'_{|\mathsf{dom}^{\texttt{mut}}(\Gamma)} \end{array}$$

Fig. 6: Typing rules

Along with promotion rules, we have two rules which make it possible to temporarily unrestrict some of the restricted variables. In the detail:

– (T-SWAP) an expression can be typed in $\Gamma; xss\ xs; ys$ if it can be typed by unrestricting some group (xs) of lent-restricted variables, by swapping this group with the current group of mutable variables (xs'). The type obtained in this way is weakened to `lent`, if it was `mut`.

– (T-UNRST) an expression can be typed in $\Gamma; xss; ys$ if it can be typed by unrestricting all strongly-restricted variables ys, provided that the type obtained in this way is `capsule` or `imm`.

We illustrate in more detail typing rules (T-CAPSULE) and (T-SWAP) for `capsule` promotion. Since in the premises of rule (T-CAPSULE) all the mutable variables in Γ are lent-restricted, the reachable graph obtained by evaluating e will contain mutable references only to local variables, which cannot be accessed from outside their scope. So the modifier of the expression can be promoted to `capsule`. Note that, if, for the expression e, we derive a type with a `lent` modifier, then the result of the evaluation of e could be an external mutable reference, therefore the value would not be a capsule.

Consider now the case in which, in the evaluation of a capsule expression, we need to perform some field assignment, as in the example of Fig.4. Typing rule (T-CAPSULE) can be applied if the initialization expression of `z` can get type `mut C` in a context with type assignment `y:mut D,z: capsule C` and the group `y` of lent-restricted variables. However, the assignment `y.f=y.f+1` is not well-typed *in this type context*, since the variable `y` has type `lent D`. However, intuitively, we can see that the assignment does not introduce any alias between `y` and the final result, since it involves only variables which are in the same group (the singleton `y`), and produces an immutable result (of type `int`). So, it should be possible to promote the expression to a capsule.

To allow such typing, we introduce rule (T-SWAP), that removes the (lent) restriction on the variables of one of the groups, say xs, so that the variables in xs can be used as mutable, adding to xss a group xs', containing all the mutable variables in Γ which are not lent-restricted yet. In this way, we know that the evaluation of the expression will not introduce any alias between the variables in the swapped group and the current group of mutable variables. Moreover, if we derive, in this new context, an immutable or capsule type, we know that the result of the expression will be a value that can be freely used. In our example, we can apply rule (T-SWAP) when deriving the type for `y.f=y.f+1`, swapping the group `y` with `x`. Instead, if we derive a mutable type, then this type is weakened to `lent`, since the result could contain references to the variables in group xs', which were lent-restricted in the original context. This is shown by the following example:

```
mut D y=new D(x1,x2);  mut x1=new A(0); mut x2 = new A(1);
capsule C z={mut A x=(y.f1=y.f2); new C(x,x)};
```

If we apply rule (T-SWAP) when deriving the type for `y.f1=y.f2`, therefore swapping the group `y` with `x`, then we derive type `mut A`, and rule (T-SWAP) would assign type `lent A` to the expression. Therefore, the declaration `mut A x=(y.f1=y.f2)` and the whole expression would be ill-typed. Indeed, the expression reduces to

```
mut D y=new D(x2,x2);  mut x1=new A(0); mut x2 = new A(1);
capsule C z={mut A x=x2; new C(x,x)};
```

in which the value of `z` is not a capsule.

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. Moreover,

$xss_{|xs}$ denotes the sequence obtained from xss by keeping (in each element of the sequence) only the variables in xs. A block is well-typed if the right-hand sides of declarations and the body are well-typed w.r.t. a type context consisting of:

- the type assignment of the enclosing scope Γ, updated by that of the locally declared variables Γ'
- groups of lent-restricted variables xss, where a locally declared (mutable) variable can belong to any group, however preserving the partition in groups of the enclosing scope, as imposed by the second side condition
- the strongly-restricted variables ys of the enclosing scope.

Typechecking a block with some local variables lent-restricted in the same group of some variables of the enclosing scope can be necessary. Consider the following example

```
mut D z=new D(0);  mut C x=new C(z,z);
capsule D y={mut D z1=new D(1);lent D z2=(x.f1=z1); new D(1)};
```

Since we need to apply the promotion to capsule to the block on right-hand side of the declarations of `y`, the context of the typing of the block must be `z : mut D, x : mut C, y : capsule D; z x;` $\emptyset$, that is, `z` and `x` are in the same group of lent-restricted variables. However, to apply rule (T-FIELD-ASSIGN) to the expression `x.f1=z1`, both `x` and `z1` have to be mutable[6]. Therefore, we have to apply rule (T-SWAP), and it must be the case that both `x` and `z1` are in the same group of lent-restricted variables. This is possible, with rule (T-BLOCK), by adding `z1` to the group `x z`, in typing the right-hand sides of the declarations.

Other rules are mostly standard, except that they model the expected behaviour of type modifiers.

In rule (T-VAR), a variable is weakened to `lent` if it belongs to some group in xss, and cannot be used at all if it belongs to ys.

In rule (T-FIELD-ACCESS), in case the field is `mut`, the type modifier of the receiver is propagated to the field. For instance, mutable fields referred through a `lent` reference are `lent` as well. If the field is immutable, instead, then the expression has type `imm`, regardless of the receiver type.

In rule (T-FIELD-ASSIGN), the receiver should be mutable, and the right-hand side must have the field type. Note that this implies the right-hand side to be either `mut` or `imm` (or of a primitive type). Hence, neither the left-hand nor the right-hand sides can be `lent` or `read`, thus preventing the introduction of aliases.

In rule (T-NEW), analogously, expressions assigned to fields cannot be `lent`. Note that an object is created with no restrictions, that is, as `mut`.

Finally, note that primitive types are 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 receiver could not.

Results The type system is sound for the operational semantics, that is, subject reduction and progress hold. Note that, since our operational model is a pure calculus, in the statements and proofs we do not need invariants on auxiliary structures such as memory.

[6] Assuming that field `f1` is mutable.

The reduction relation $e \longrightarrow e'$ is defined in the full paper [8]. We write $\vdash e : T$ for $\emptyset;\emptyset;\emptyset \vdash e : T$.

Theorem 1. *Let $\vdash e : T$. Then, either e is a value or $e \longrightarrow e'$ for some e' such that $\vdash e' : T$ is derivable.*

In addition to the standard soundness property, the `capsule` and `imm` modifiers have the expected behaviour, that is, in the evaluation of a well-typed expression,

- initialization expressions of `capsule` variables reduce to values whose *free variables are immutable*, and
- values associated to immutable variables *are not modified by the evaluation.*

The formalization and proof of these properties can be found in the full paper [8].

4 Related work and conclusion

A first, informal, version of our type modifiers has been presented in [13]. In [14] were introduced the `capsule` and `lent` modifiers, and a preliminary version of the promotion from mutable to capsule. The main novelties w.r.t. [14] are the introduction of the immutable and readable modifiers, the readable to immutable promotion, the formalization of the type system, and the proof of its properties.

Our type system combines in a novel and powerful way different features existing in previous work. Notably, the `capsule` notion has many variants in the literature, such as *external uniqueness* [5], *balloon* [1,13], *island* [7], *recovery* [9], and the fact that aliasing can be controlled by using *lent* (*borrowed*) references is well-known [11].

However, in our type system the `lent` notion is used in a novel way to achieve capsule promotion, since external mutable references are not forbidden once and for all as in recovery [9] but only lent-restricted, as illustrated by the last example in Sect.2.

Moreover, uniqueness is guaranteed by linearity, that is, by allowing at most one use of a `capsule` reference, rather than by destructive reads as in [9,3]. Destructive reads allows *isolated* fields, but has a serious drawback: an *isolated* field can become unexpectedly not available, hence any object contract involving such field can be broken.

Our `read` modifier is different from *readonly* as used, e.g., in [2]. An object cannot be modified through a readable/readonly reference. However, `read` also prevents static aliasing.

Javari [15] is a working backward-compatible extension of Java which also supports the *readonly* type modifier, and makes a huge design effort to support *assignable* and *mutable* fields, to have fine-grained readonly constraints. The need of such flexibility is motivated by performance reasons. In our design philosophy, we do not offer any way to a programmer of breaking the invariants enforced by the type system. Since our invariants are very strong, we expect compilers to

be able to perform optimization, thus recovering most of the efficiency lost to properly use immutable and readable objects.

Finally, the Pony language [6], providing an implementation as well, builds on the capabilities/recovery mechanisms of [9] as we do, but goes in a different direction, by distinguishing six different reference capabilities: *isolated* (similar to our capsule, but allowed in fields with destructive reads); *value* (similar to our immutable); *reference* (similar to our mutable); *box* (similar to readonly); *tag* (only allows object identity checks) and finally *transition* (a subtype of *box* that can be converted in *value*: a way to create values without using isolated references).

The key contributions of the paper are:

- A powerful type system for tracing mutation and aliasing in class-based imperative languages, providing: type modifiers for restricting the use of references; rules for promoting references to a less restrictive type at the price of restricting other references; rules for temporarily unrestricting references for typing subexpressions.
- A non standard operational model of the language as a pure calculus, relying on the language ability to represent cyclic object graphs.

This work is part of the development of L42, a novel programming language designed to support massive use of libraries, see the web site `L42.is`. The current L42 prototype is important as proof-of-evidence that the type system presented in this paper can be smoothly integrated with features of a realistic language. The prototype implements an algorithmic version of the type system which, roughly, attempts at applying promotions in all the possible ways, which are finitely many and, in practice, very few. A theoretical counterpart of such algorithmic version will be subject of further work. As a long term goal, we also plan to investigate (a form of) Hoare logic on top of our model. Finally, it should be possible to use our approach to enforce safe parallelism, on the lines of [9,13].

Acknowledgments. We are grateful to the anonymous reviewers for their useful suggestions, which led to substantial improvements.

References

1. Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP'97* , volume 1241 of *LNCS*, pages 32–59. Springer.
2. Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *OOPSLA 2004*, pages 35–49. ACM Press.
3. John Boyland. Semantics of fractional permissions with nesting. *ACM TOPLAS*, 32(6), 2010.
4. Andrea Capriccioli, Marco Servetto, and Elena Zucca. An imperative pure calculus. In *ICTCS'15*, ENTCS 322, 87–102, 2015.
5. David Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP'03*, volume 2473 of *LNCS*, pages 176–200. Springer.
6. Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *AGERE! 2015*, pages 1–12. ACM Press.

7. Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP'07*, volume 4609 of *LNCS*, pages 28–53. Springer.
8. Paola Giannini, Marco Servetto, and Elena Zucca. Transparent aliasing and mutation control. `http://people.unipmn.it/giannini/papers/ICTCS16-complete.pdf`.
9. Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *OOPSLA 2012*, pages 21–40. ACM Press.
10. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
11. Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *POPL 2012*, pages 557–570. ACM Press.
12. Marco Servetto and Lindsay Groves. True small-step reduction for imperative object-oriented languages. In *FTfJP'13*, pages 1–7. ACM Press.
13. Marco Servetto, David J. Pearce, Lindsay Groves, and Alex Potanin. Balloon types for safe parallelisation over arbitrary object graphs. In *WoDet 2013*.
14. Marco Servetto and Elena Zucca. Aliasing control in an imperative pure calculus. In *APLAS 15*, volume 9458 of *LNCS*, pages 208–228. Springer.
15. Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA 2005*, pages 211–230. ACM Press.
16. Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic Java. In *OOPSLA 2010*, pages 598–617. ACM Press.