Tracing and Preventing Sharing and Mutation

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

[1] Computer Science Institute, DiSIT, Università del Piemonte Orientale, Italy
paola.giannini@uniupo.it
[2] School of Engineering and Computer Science,Victoria University of Wellington, New Zealand
servetto@ecs.vuw.ac.nz
[3] DIBRIS, Università di Genova, Italy
elena.zucca@unige.it

Abstract. We present a type and effect system for tracing and preventing sharing and mutation in imperative languages. That is, on one hand, the type system *traces* sharing possibly introduced by the evaluation of an expression, so that uniqueness and immutability properties can be easily detected. On the other hand, sharing and mutation can be *prevented* by *type qualifiers* which forbid some actions. Sharing is directly represented at the syntactic level as a relation among free variables, thanks to the fact that in the underlying calculus memory is encoded in terms.

Keywords: type inference · sharing · effects

1 Introduction

The last few decades have seen considerable interest in type systems for controlling sharing and interference, to make programs easier to maintain and understand. A simple and widely used technique is to enrich the type of an expression evaluating to a reference x by *type qualifiers* [29,18,25,10] or by *capabilities* [5,7]. Depending on the qualifier of x, restrictions are imposed and assumptions can be made on the (reachable) object graph of x. In this paper, we consider a small yet powerful set of qualifiers with the meaning described below.

If x is *mutable* (`mut`), then no restrictions are imposed and no assumptions can be made. Restrictions are imposed by the following modifiers:

– If x is *read-only*, then fields cannot be modified ($x.f{=}e$ is not legal).
– If x is *lent* [28,14,17], also called *borrowed* in literature [4,25], then the object graph of x can be manipulated, but not shared, by a client.
– The two modifiers can be combined so that neither modification nor sharing are permitted. That is, both the *read-only* and the *lent* restriction are imposed; this modifier was called *readable* in [17].

In the following formalization, these three qualifiers will be denoted `read`, $\mathtt{mut}^{\mathtt{lent}}$, and $\mathtt{read}^{\mathtt{lent}}$, respectively. Note that they *do not allow* any assumption on the object graph. For instance, the object graph of a `read` reference could be modified through other references, and connections could be added to the object graph of a $\mathtt{mut}^{\mathtt{lent}}$ reference through other references.

To be able to make assumptions on the object graph of a reference, the key notion is expressed by the `caps` qualifier. If x is `caps`, then a client can assume that this subgraph is an isolated portion of store, that is, all its (non immutable) nodes can be reached only through this reference. We use the name *capsule* for this property, to avoid confusion with many variants in literature [9,1,27,19,11,18]. If x is `caps`, and, moreover, is `read`, then it is *immutable* (`imm`). That is, $x.f{=}e$ is not legal, and, moreover, we can assume that the object graph of x will not be modified through any other reference.

(Variants of) such qualifiers have appeared in previous literature, and, in particular, they are all smoothly integrated in [17]. However, in [17] the capsule and immutability property were detected by a rather complex type system, based on the *recovery* technique, firstly introduced in [18,10]. In this paper, instead, such properties are naturally detected by a type and effect system which *traces sharing*: that is, given an expression e with free variables, computes a *sharing relation* $\mathcal{S}$ on such free variables, plus a distinguished variable `res` denoting the result. The fact that two variables, say x and y, are in the same equivalence class in $\mathcal{S}$, means that the evaluation of e can possibly introduce sharing between x and y, that is, connect their object graphs, so that a modification of (a subobject of) x could affect y as well, and conversely.

For instance, given the expression $x\mathtt{.}f\mathtt{=}y\mathtt{;}z\mathtt{.}f$, its evaluation introduces connections between x and y, and between `res` (the result) and z. In this way, an expression is a capsule if its result will be disjoint from any free variable (formally, `res` is a singleton in $\mathcal{S}$). For instance, the expression $x\mathtt{.}f\mathtt{=}y\mathtt{;new}\ C\mathtt{(new}\ D\mathtt{()).}f$ is a capsule, whereas the previous expression is not.

Tracing sharing has been firstly used in [13] to detect capsule and in [15] also immutability properties. In this paper, this technique is smoothly integrated with qualifiers which *prevent* sharing and mutation, providing a very expressive type system.

We adopt an execution model [26,6,28] where memory is encoded in the language itself, making possible to express uniqueness and immutability properties in a simple and direct way. In this paper, for lack of space, the calculus is only informally presented.

The rest of the paper is organized as follows: in Sect.2 we informally present the type system and illustrate its expressive power by examples. In Sect.3 we formalize the type and effect system, and in Sect.4 we state some of its properties. Finally, in Sect.5 we discuss related and further work. A formal presentation of the operational semantics on which the results of Sect.4 rely can be found in a companion technical report [16].

2 Language and examples

The type system is presented on top of a toy language with an object-oriented flavour, inspired by Featherweight Java [20].

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 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. The syntax of the language is given below.

cd	$::= \texttt{class}\ C\ \{fds\ mds\}$	class declaration
fd	$::= \texttt{imm}\,C\,f\texttt{;} \mid \texttt{mut}\,C\,f\texttt{;} \mid \texttt{read}\,C\,f\texttt{;}$	field declaration
md	$::= T\ m\ \texttt{(}q^\tau, T_1\,x_1, \ldots, T_n\,x_n\texttt{)}\ \{e\}$	method declaration
e	$::= x \mid e\texttt{.}f \mid e\texttt{.}f\texttt{=}e' \mid \texttt{new}\ C\texttt{(}es\texttt{)} \mid \{ds\ e\} \mid e\texttt{.}m\texttt{(}es\texttt{)}$	expression
d	$::= T\,x\texttt{=}e\texttt{;}$	declaration
T	$::= q^\tau\,C$	type
q	$::= \texttt{mut} \mid \texttt{read} \mid \texttt{imm} \mid \texttt{caps}$	qualifier
τ	$::= \epsilon \mid \texttt{lent}$	(optional) lent tag

In method declarations there is an additional component, the type qualifier for `this`, written as first element of the parameter list.

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.

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 block consisting of a sequence of local declarations and a body, or a method invocation. A declaration specifies a type, a variable and an initialization expression. We assume no multiple declarations of variables in a block. A type consists of a class name and a qualifier.

As sketched in the Introduction, depending on the qualifier of a reference x, restrictions are imposed and assumptions can be made on the object graph of x.
If x is *mutable* (`mut`), then no restrictions are imposed and no assumptions can be made.
If x is *readonly* (`read`), then fields cannot be modified ($x.f{=}e$ is not legal).
If x is *immutable* (`imm`), then it is `read`, that is, $x.f{=}e$ is not legal, and, moreover, we can assume that the *object graph* of x will not be modified through any other reference. As a consequence, an immutable reference can be safely shared in a multithreaded environment.
If x is *capsule* (`caps`), then we can assume that the object graph of x is an isolated portion of store, that is, all its (non immutable) nodes can be reached only through this reference. Capsule expressions can initialize both mutable and immutable references. If a capsule is assigned to a mutable reference y, then y can rely on the fact that no part of this subgraph can be updated through another reference. This allows programmers (and static analysis) to identify mutable state that can be safely handled by a thread. To preserve the capsule property,

we need an *affinity constraint* which, in our case, can be simply expressed as a syntactic well-formedness condition, rather than by context rules, as in linear logic-style type systems: in well-formed expressions capsule references can occur at most once in their scope.
Qualifiers can be optionally tagged `lent`. This imposes the additional constraint that the object graph cannot be shared by a client. That is, the object graph of x cannot be stored in a previously disjoint object graph. In particular $x.f{=}x$ is allowed, whereas $z.f{=}x$ is not. This tag makes sense only for `mut` and `read` qualifiers, since `imm` references can be freely shared and and `caps` references are temporary. According to the substitution principle we have that the subtyping relation is the reflexive and transitive relation on types induced by:

$$\texttt{caps} \leq \texttt{mut} \leq \texttt{read} \qquad \texttt{caps} \leq \texttt{imm} \leq \texttt{read} \qquad \epsilon \leq \texttt{lent}$$

Examples We illustrate now the use of the qualifiers by some examples and show how they can express several ownership properties, see [8]. We assume `mut` as default qualifier and, for sake of readability, we use a Java-like syntax with additional constructs, such static methods, private fields, etc. Consider the following example in conventional Java, modelling a graph with a list of nodes, and a constructor taking in input such list

```
class Graph{
  private final NodeList nodes;
  private Graph(NodeList nodes){this.nodes=nodes;}

  static Graph factory(NodeList nodes){
    return new Graph(nodes.deepClone());
  }
}
```

and assume that we want to ensure that the list of nodes of a graph is not referred from the external environment (that is, the graph is the *owner* of its list of nodes). Without a type system for aliasing control, as shown, the factory method should deeply clone the argument. This solution, called *defensive cloning* [3], is very popular in the Java community, but inefficient, since it requires to duplicate the object graph of the parameter, until immutable nodes are reached.

With our type system, instead, we may require the parameter of the `factory` method to be a `caps`:

```
class Graph{ ...
  static Graph factory(caps NodeList nodes){
    return new Graph(nodes);
   }
```

In this way, the factory method *moves* an isolated portion of store as local store of the newly created object. Cloning, if needed, becomes responsibility of the client which provides the list of nodes to the graph. In other words, the capsule notion models an efficient *ownership transfer*[4]. That is, in classical ownership systems the property that y is "owned" by x holds forever, whereas the capsule notion is more dynamic: a capsule can be "opened", that is, assigned to a standard reference and modified, and then we can recover the original capsule guarantee (in the example, `new Graph(nodes)` is a capsule).

Depending on how we expose the owned data, we can finely tune the way they can be manipulated by clients. Different options, and their combinations, may be appropriate in different circumstances. Consider the following ways in which we can access the `NodeList` of a `Graph`.

```
class Graph{ ...
 read NodeList readNodes(read){return this.nodes;}//(1)

 mut^lent NodeList borrowNodes(mut^lent){return this.nodes;}//(2)

 read^lent NodeList getNodes(read^lent){return this.nodes;}//(3)

 caps NodeList copyNodes(read^lent){return nodes.deepClone();}//(4)
}
```

(1) If the list of nodes is returned `read`, then the client code is allowed to get a permanent reference to the internal data, and to track such data changing over time. However, it is prevented to mutate the data, so multi-object invariants on such data should be safe. This closely model the *owners-as-modifiers* pattern.
(2) If the list of nodes is returned `mut`$^{\texttt{lent}}$, then client code is allowed to get a temporary reference to the internal data, and mutate it. However, the client cannot store such data, and local reasoning can be used to track the lifetime of the temporary reference. For example (`ROG` stands for "reachable object graph"):

```
EvilCode evil=new EvilCode();
...
Graph g=Graph.factory(...);
//g has control of its ROG here
evil.attack(g.borrowNodes());
//g has again control of its ROG
//ROG(g) and ROG(evil) are disjoint
```

(3) This is the most conservative and efficient option: The user can read the data, and the lifetime of such `read`$^{\texttt{lent}}$ references can be tracked.
In our opinion, in most cases it would be a good software development practice to use this qualifier for getters over mutable data.
(4) This solution models the *owners-as-dominators* pattern. In the class `NodeList` the method `deepClone` could have the following declaration:

```
 caps NodeList deepClone(read^lent){ ... }
```

[4] Other work in literature supports ownership transfer, for example [24,9]. However, it is generally applied to uniquess/external uniqueness, thus not the whole object graph is transferred.

In this way, the client has no access to the internal data. This requires duplication, but, with respect to conventional ownership, it is more efficient when the result is used to initialize a new graph:

```
Graph.factory(oldGraph.copyNodes())
```

calls a single deep clone operation in our approach, while the equivalent plain Java approach would require to clone the `ROG` twice.

In our approach all properties are *deep*, that is, propagate to the whole object graph. Instead, most ownership approaches allows one to distinguish subparts of the object graph that are referred but not logically owned. This choice has some advantages, for example the Rust language[5] leverages on ownership to control object deallocation without a garbage collector [21]. However, in most ownership based approaches it is not trivial to encode the concept of full encapsulation, while supporting (open) subtyping and avoiding defensive cloning.

3 Type system

We introduce now the type and effect system for the language.

A *sharing relation* $\mathcal{S}$ is an equivalence relation on variables. As usual $[x]_{\mathcal{S}}$ denotes the *equivalence class of x in $\mathcal{S}$*. We will call *connections* the elements $\langle x, y\rangle$ of a sharing relation, and say that x and y are *connected*. The intuitive meaning is that, if x and y are connected, then their object graphs in the store are possibly shared (that is, not disjoint), hence a modification of the object graph of x could affect y as well, and conversely.

The typing judgment has shape

$$\Gamma \vdash e : C \,|\, \mathcal{S}$$

where Γ is a *type environment*, that is, an assignment of types to variables, written $x_1{:}T_1, .., x_1{:}T_n$, and $\mathcal{S}$ is a sharing relation on the (non immutable) free variables in e, plus a distinguished variable `res` denoting the result of e. The intuitive meaning is that $\mathcal{S}$ represents the connections possibly introduced by the evaluation of e, and, in particular, the variables in $[\texttt{res}]_{\mathcal{S}}$ are the ones that will be possibly connected to the result of the expression.

We write $\mathsf{capsule}(\mathcal{S})$ if $[\texttt{res}]_{\mathcal{S}}$ is a singleton ($[\texttt{res}]_{\mathcal{S}} = \{\texttt{res}\}$.) In this case, the expression e denotes a *capsule*, that is, reduces to a portion of store which is isolated, except for immutable references.

The class table is abstractly modelled by the following functions:

- $\mathsf{fields}(C)$ gives, for each declared class C, the sequence of its field declarations $T_1\, f_1\,;..T_n\, f_n\,;$.
- $\mathsf{meth}(C, m)$ gives, for each method m declared in class C, the tuple $\langle T \,|\, \mathcal{S}, q^{\tau}, T_1\, x_1, \ldots, T_n\, x_n, e\rangle$ consisting of its return type T and sharing effects $\mathcal{S}$, qualifier for `this`, parameters, and body.

We assume a well-typed class table, that is, method bodies are expected to be well-typed with respect to method types. Formally,
if $\mathsf{meth}(C, m) = \langle T \,|\, \mathcal{S}, q^{\tau}, T_1\, x_1..T_n\, x_n, e\rangle$, then

[5] `rust-lang.org`

– $\Gamma \vdash e : T \,|\, \mathcal{S}$, with $\Gamma = \texttt{this}{:}q^{\tau}\, C, x_1{:}T_1, .., x_n{:}T_n$.

The typing rules are given in Fig.1. For sharing relations we use the following notations where X denotes a set of variables.

– A sequence of mutually disjoint subsets of X, say $X_1 \cdots X_n$, represents the smallest equivalence relation on X containing the connections $\langle x, y\rangle$, for x and y belonging to the same X_i. So, ϵ represents the identity relation on any set of variables. Note that this representation is deliberately ambiguous as to the domain of the defined equivalence.
– $\mathcal{S}_1 + \mathcal{S}_2$ is the smallest equivalence relation containing $\mathcal{S}_1$ and $\mathcal{S}_2$. It is easy to show that sum is commutative and associative.
– $\mathcal{S}\backslash X$ is obtained by "removing" the variables in X from $\mathcal{S}$, that is, is the smallest equivalence relation containing the connections $\langle x, y\rangle$, for all $\langle x, y\rangle \in \mathcal{S}$, such that $x \notin X$ and $y \notin X$.
– $\mathcal{S}\backslash\texttt{res}$ coincides with $\mathcal{S}$ except for `res` which is no longer connected to any variable, that is, it contains all $\langle x, y\rangle$ such that either $x = y = \texttt{res}$ or $\langle x, y\rangle \in \mathcal{S}$ and $x \neq \texttt{res}$ and $y \neq \texttt{res}$.
– $\mathcal{S}[y/x]$ is obtained by "replacing" x by y in $\mathcal{S}$, that is, is the smallest equivalence relation containing the connections:
  $\langle z, z'\rangle$, for all $\langle z, z'\rangle \in \mathcal{S}$, $z \neq x, z' \neq x$
  $\langle y, z\rangle$, for all $\langle x, z\rangle \in \mathcal{S}$.
– $\mathcal{S}_1$ *has less (or equal) sharing effects than* $\mathcal{S}_2$, dubbed $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$, if, for all x, $[x]_{\mathcal{S}_1} \subseteq [x]_{\mathcal{S}_2}$.

In rule (T-VAR), the evaluation of a (not immutable nor capsule) variable connects the result of the expression with the variable itself. Note that, given our notation, $\{x, \texttt{res}\}$ represents a sharing relation in which the only non trivial connections is between x and `res`. In rule (T-IMM-VAR), the evaluation of an immutable or capsule variable does not introduce any connection, so the resulting sharing relation is the identity relation.

Rule (T-SUB) is the usual subsumption.

In rule (T-FIELD-ACCESS), in case the field is `mut`, the qualifier of the receiver is propagated to the field. For instance, mutable fields referred through an `imm` reference are `imm` as well. If the field is `read`, and the tag of the receiver is `lent`, then it is propagated to the field. Otherwise, the expression has the field type, regardless of the receiver type. Note that, if the field is `read` and the receiver is `imm`, the field access can be typed `imm` as well by promotion. The connections introduced by a field access are those introduced by the evaluation e. Since $[\texttt{res}]_{\mathcal{S}}$ contains all the references that could be in the object graph of the result of e, it also contains all the references that could be in the object graph of $e.f$. However, in case the field has a `imm` qualifier, since the `imm` property is deep, then the result of the expression is not connected to any mutable or readable reference.

In rule (T-FIELD-ASSIGN), the receiver should be mutable, and the right-hand side should have the field type. The sharing effects of a field assignment are (the sum of) those of the two expressions ($\mathcal{S}_1$ and $\mathcal{S}_2$). Moreover, if the receiver is `lent`,

$$\textsc{(t-var)}\ \frac{}{\Gamma \vdash x : q^{\tau}\,C \mid \{x, \mathtt{res}\}}\ \begin{array}{l}\Gamma(x) = q^{\tau}\,C\\ q \not\leq \mathtt{imm}\end{array} \qquad \textsc{(t-imm-var)}\ \frac{}{\Gamma \vdash x : q\,C \mid \epsilon}\ \begin{array}{l}\Gamma(x) = q\,C\\ q \leq \mathtt{imm}\end{array}$$

$$\textsc{(t-sub)}\ \frac{\Gamma \vdash e : q^{\tau}\,C \mid \mathcal{S}}{\Gamma \vdash e : q'^{\tau'}\,C \mid \mathcal{S}}\ \begin{array}{l}q \leq q'\\ \tau \leq \tau'\end{array} \qquad \textsc{(t-field-access)}\ \frac{\Gamma \vdash e : q^{\tau}\,C \mid \mathcal{S}}{\Gamma \vdash e\,.f_i : q'^{\tau'}\,C_i \mid \mathcal{S}'}\ \begin{array}{l}\mathsf{fields}(C) = T_1\,f_1\,;\ \ldots\ T_n\,f_n\,;\\ i \in 1..n,\ T_i = q_i\,C_i\\ q'^{\tau'}, \mathcal{S}' = \begin{cases} q^{\tau}, \mathcal{S} & \text{if } q_i = \mathtt{mut}\\ q_i^{\tau}, \mathcal{S} & \text{if } q_i = \mathtt{read}\\ q_i, \mathcal{S}\backslash\mathtt{res} & \text{if } q_i = \mathtt{imm}\end{cases}\end{array}$$

$$\textsc{(t-field-assign)}\ \frac{\Gamma \vdash e_1 : \mathtt{mut}^{\tau_1}\,C \mid \mathcal{S}_1 \qquad \Gamma \vdash e_2 : q_i^{\tau_2}\,C_i \mid \mathcal{S}_2}{\Gamma \vdash e_1\,.f_i\mathtt{=}e_2 : q_i^{\tau}\,C_i \mid \mathcal{S}'}\ \begin{array}{l}\mathsf{fields}(C) = T_1\,f_1\,;\ \ldots\ T_n\,f_n\,;\\ i \in 1..n,\ T_i = q_i\,C_i\\ \langle \tau, \mathcal{S}\rangle = \langle \tau_1, \mathcal{S}_1\rangle + \langle \tau_2, \mathcal{S}_2\rangle\\ \mathcal{S}' = \begin{cases} \mathcal{S}\backslash\mathtt{res} & \text{if } q_i = \mathtt{imm}\\ \mathcal{S} & \text{otherwise}\end{cases}\end{array}$$

$$\textsc{(t-new)}\ \frac{\Gamma \vdash e_i : q_i^{\tau_i}\,C_i \mid \mathcal{S}_i \qquad \forall i \in 1..n}{\Gamma \vdash \mathtt{new}\ C\mathtt{(}e_1, \ldots, e_n\mathtt{)} : \mathtt{mut}^{\tau}\,C \mid \mathcal{S}}\ \begin{array}{l}\mathsf{fields}(C) = q_1\,C_1\,f_1\,;\ \ldots\ q_n\,C_n\,f_n\,;\\ \langle \tau, \mathcal{S}\rangle = \sum_{i=1}^{n} \langle \tau_i, \mathcal{S}_i\rangle\end{array}$$

$$\textsc{(t-block)}\ \frac{\begin{array}{c}\Gamma[\Gamma'] \vdash e_i : T_i \mid \mathcal{S}_i\ \ 1 \leq i \leq n\\ \Gamma[\Gamma'] \vdash e : T \mid \mathcal{S}'\end{array}}{\Gamma \vdash \{\,T_1\,x_1\mathtt{=}e_1\,\mathtt{;}..\,T_n\,x_n\mathtt{=}e_n\,\mathtt{;}\ e\} : T \mid \mathcal{S}\backslash\mathsf{dom}(\Gamma')}\ \begin{array}{l}\Gamma' = x_1{:}T_1, .., x_n{:}T_n\\ \mathcal{S}'_i = \mathcal{S}_i[x_i/\mathtt{res}]\\ \mathcal{S} = \sum_{i=1}^{n} \mathcal{S}'_i + \mathcal{S}'\end{array}$$

$$\textsc{(t-invk)}\ \frac{\Gamma \vdash e_i : T_i \mid \mathcal{S}_i \qquad \forall i \in 0..n}{\Gamma \vdash e_0\,.\,m\mathtt{(}e_1, .., e_n\mathtt{)} : T \mid \mathcal{S}\backslash\{\mathtt{this}, x_1, .., x_n\}}\ \begin{array}{l}T_0 = q^{\tau}\,C\\ \mathsf{meth}(C, m) = \langle T \mid \mathcal{S}', q^{\tau}, T_1\,x_1, \ldots, T_n\,x_n, e\rangle\\ \mathcal{S}'_0 = \mathcal{S}_0[\mathtt{this}/\mathtt{res}] \qquad \mathcal{S}'_i = \mathcal{S}_i[x_i/\mathtt{res}]\\ \mathcal{S} = \sum_{i=0}^{n} \mathcal{S}'_i + \mathcal{S}'\end{array}$$

$$\textsc{(t-caps)}\ \frac{\Gamma \vdash e : \mathtt{mut}\,C \mid \mathcal{S}}{\Gamma \vdash e : \mathtt{caps}\,C \mid \mathcal{S}}\ \mathsf{capsule}(\mathcal{S}) \qquad \textsc{(t-imm)}\ \frac{\Gamma \vdash e : \mathtt{read}^{\tau}\,C \mid \mathcal{S}}{\Gamma \vdash e : \mathtt{imm}\,C \mid \mathcal{S}}\ \mathsf{capsule}(\mathcal{S})$$

Fig. 1. Type system

then the constraint holds that its connections cannot be augmented, hence the equivalence class of the result of the right-hand side ($[\mathtt{res}]_{\mathcal{S}_2}$) should be included in the one of the receiver ($[\mathtt{res}]_{\mathcal{S}_1}$). The converse holds if the right-hand side is `lent` (hence if both are `lent` then $[\mathtt{res}]_{\mathcal{S}_1} = [\mathtt{res}]_{\mathcal{S}_2}$). In either case, the result of the assignment is `lent` as well. Formally, here and in rule (t-new), the notation $\langle \tau, \mathcal{S}\rangle = \sum_{i=1}^{n} \langle \tau_i, \mathcal{S}_i\rangle$ is defined as follows:

- for each $i \in 1..n$, if $\tau_i = \mathtt{lent}$, then it must be $[\mathtt{res}]_{\mathcal{S}_j} \subseteq [\mathtt{res}]_{\mathcal{S}_i}$, for all $j \in 1..n$
- if this condition is violated for some $i \in 1..n$, then the notation is undefined; otherwise, $\mathcal{S} = \sum_{i=1}^{n} \mathcal{S}_i$, and $\tau = \mathtt{lent}$ if $\tau_i = \mathtt{lent}$ for some $i \in 1..n$.

An assignment expression will reduce to the value of the expression on its right-side, therefore the connections of its result are as for rule (t-field-access). Note that, immutable or read-only fields can be assigned, since the qualifier asserts the immutability or read-only property of the object referred to not of the field itself.

In rule (T-NEW), an object is created with no restrictions, that is, as `mut`. The sharing effects of a constructor invocation are (the sum of) those of the arguments. Note that the equivalence class of `res` in the sum of the sharing relations is the union of the equivalence classes of `res` in the summed sharing relations. Indeed the object created is connected to its fields. However, since we can prove that *the sharing relation $\mathcal{S}$ associated to expression having the* `imm` *qualifier is such that* $[\texttt{res}]_{\mathcal{S}} = \{\texttt{res}\}$, `imm` fields are not connected to the result of the constructor. Moreover, analogously to rule (T-FIELD-ASSIGN), if one argument is `lent`, then its sharing effects cannot be augmented, and the created object is `lent` as well.

In rule (T-BLOCK), the initialization expressions and the body of the block are typechecked in the current type environment, enriched by the association to local variables of their declaration types. We denote by $\Gamma[\Gamma']$ the type environment which is equal to Γ' on the variables where Γ' is defined, to Γ otherwise. The connections introduced by a block are obtained modifying those introduced by the evaluation of the initialization expressions ($\mathcal{S}_i, 1 \leq i \leq n$) plus those introduced by the evaluation of the body $\mathcal{S}'$. More precisely, for each declared variable, the connections of the result of the initialization expression are transformed in connections to the variable itself. Finally, we remove from the resulting sharing relation the local variables.

In rule (T-INVK), the typing of e_0 `.` m`(`$e_1, .., e_n$`)` is similar to the typing of the block `{` T_0 `this=`e_0`;` T_1 x_1`=`e_1`;`.. T_n x_n`=`e_n`;` e`}` However, while in a block local variable declarations can refer to each other, the receiver e_0 and the arguments e_i ($1 \leq i \leq n$) do not refer to `this` and the formal parameters, hence the sharing effects among them are only those caused by the method body e.

In some cases it is possible to move the type of an expression against the subtype hierarchy, that is, to *promote* an expression. A `mut` expression can be promoted to `caps`, rule (T-CAPS), when its result will not be connected to external non immutable references. For example, consider the following example, where we use integers but any immutable reference could be used instead

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

The initialization expression for `z` can be given type `capsule` by using rule (T-CAPS) since the result of the block is not connected to any external variable and the block has type `mut C`. Note that in rule (T-CAPS), expression e cannot be tagged `lent`. Consider the following variation of the previous example

mut D y=new D(0); ??? C z={mut$^{\texttt{lent}}$ D x=new D(y.f); new C(x,x)};

Also in this case the result of the block is not connected to any external variable. However, the block has type mut$^{\texttt{lent}}$C. If we could use rule (T-CAPS) to promote to type `caps` by subtyping the block expression would have type `mut` and so `???` could be `mut`, which is not correct.
A `read` expression can be promoted to `imm`, rule (T-IMM), when its result will not

be connected to external non immutable references. In this case the expression could be tagged `lent`. For example

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

is typable by deriving type $\texttt{mut}^{\texttt{lent}}\texttt{C}$ for the block, applying the subtyping to get $\texttt{read}^{\texttt{lent}}\texttt{C}$ and then using rule (T-IMM). So we can correctly derive type `imm C` for the block.

4 Results

In this section we present the main formal results on our calculus.
We start by stating that if a variable is declared with the lent modifier, then the evaluation of the expressions in its scope do not increase its connections.

Theorem 1 (Typing Lent). *Let* $\Gamma \vdash \{T_1\, x_1\texttt{=}e_1\texttt{;}..T_n\, x_n\texttt{=}e_n\texttt{;}\ e\} : T \,|\, \mathcal{S}\backslash\mathsf{dom}(\Gamma')$ *where* $\Gamma' = x_1{:}T_1, .., x_n{:}T_n$, $\Gamma[\Gamma'] \vdash e_i : T_i \,|\, \mathcal{S}_i$, $\Gamma[\Gamma'] \vdash e : T \,|\, \mathcal{S}'$, $\mathcal{S}'_i = \mathcal{S}_i[x_i/\textsf{res}]$ *and* $\mathcal{S} = \sum\limits_{i=1}^{n} \mathcal{S}'_i + \mathcal{S}'$. *Then,* $T_i = q^{\textit{lent}}C$ *implies* $[x_i]_{\mathcal{S}} = [x_i]_{\mathcal{S}'_i}$.

The other results state properties of the type system with respect to the operational semantics, which is reported in a companion technical report [16]. Here we provide a minimal presentation, in order to make the results understandable.

In the operational semantics we use variable declarations to directly represent the store. That is, a declared (non capsule) 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 [2,22]. Semantics is defined by a *congruence* relation, which captures structural equivalence, and a *reduction* relation, $\longrightarrow$,which models actual computation, similarly to what happens, e.g., in π-calculus [23].

A *value* is the result of the reduction of an expression, and is either a variable (a reference to an object), or a block where the declarations are evaluated (hence, correspond to a local store) and the body is in turn a value, or a constructor call where argument are evaluated. 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 reference. Capsule references are not part of the store. They are used as a temporary reference initialized with an isolated portion of store to be"moved" to another location in the store, without introducing sharing. In the operational semantic, a declaration of a variable x whose type has the `caps` qualifier, when the initialisation expression is reduced to a value, is eliminated by substituting the occurrence of the variable with its value.

v	$::= x \mid \texttt{new}\ C\texttt{(}vs\texttt{)} \mid \{dvs\ x\} \mid \{dvs\ \texttt{new}\ C\texttt{(}vs\texttt{)}\}$	value
dv	$::= q^{\tau}\, C\, x\texttt{=}rv\texttt{;} \qquad q \neq \texttt{caps}$	evaluated declaration
rv	$::= \texttt{new}\ C\texttt{(}xs\texttt{)} \mid \{dvs\ x\} \mid \{dvs\ \texttt{new}\ C\texttt{(}xs\texttt{)}\}$	right-value

The soundness of the type system for the operational semantics says that in addition to preserving the type of expressions reduction also produces an expression that has less sharing.

Theorem 2 (Subject reduction). *If* $\Gamma \vdash e : T \,|\, \mathcal{S}$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' : T' \,|\, \mathcal{S}'$ *where* $T' \leq T$ *and* $\mathcal{S}' \sqsubseteq \mathcal{S}$.

In the following with $\vdash e$ we mean that $\vdash e : T \mid \mathcal{S}$ for some T and $\mathcal{S}$ and since in this case e is closed $\mathcal{S}$ can only be the identity sharing relation.

The following expresses the standard progress property.

Theorem 3 (Progress). *$\vdash e$ and e not a value implies $e \longrightarrow e'$ for some e'.*

In addition to preserving the type of expressions, reduction also preserves the immutable and capsule properties of subexpressions.

To trace the expression associated to a variable x in a store we assume that there is no shadowing and define *contexts that have a hole on the right-hand-side of the (unique) declaration of x* by:

$$\mathcal{D}_x ::= \{ds\ \ T\,x\texttt{=}[\ ]\texttt{;}\ \ ds'\ e\} \mid \{ds\ \ T\,y\texttt{=}\mathcal{D}_x\texttt{;}\ \ ds'\ e\} \mid \{ds\ \mathcal{D}_x\} \mid \mathcal{D}_x\texttt{.}f \\ \mid \mathcal{D}_x\texttt{.}f\texttt{=}e \mid e\texttt{.}f\texttt{=}\mathcal{D}_x \mid \texttt{new}\ C\texttt{(}es\,\mathcal{D}_x\,es'\texttt{)} \mid \mathcal{D}_x\texttt{.}m\texttt{(}es\texttt{)} \mid e\texttt{.}m\texttt{(}es\,\mathcal{D}_x\,es'\texttt{)}$$

We use the notations $\mathcal{D}_{qx}$ to refer to a declaration with a specific qualifier and $\mathsf{type}(y, \mathcal{D}_x) = T$ if $T\,y\texttt{=}e\texttt{;}$, for some e, is a declaration in one of the blocks enclosing the hole of $\mathcal{D}_x$.

We can now state that the promotion rules for capsule and immutable are sound w.r.t. the operational semantics, i.e., once their initialisation expression is evaluated, variables declared with `caps` modifier refers to isolated portion of the store and variables declared with `imm` modifier are not modified by execution of expressions in their scope. To formulate the isolation property for capsule, given a right value rv consider $\mathsf{gc}(rv)$ to be obtained by rv removing in blocks the declarations which are not reachable from the body.

Theorem 4 (Capsule and Immutable). *If $\vdash \mathcal{D}_{qx}[e]$ and $\mathcal{D}_{qx}[e] \longrightarrow^{\star} \mathcal{D}'_{qx}[rv]$ with $q = \texttt{caps}$ or $q = \texttt{imm}$, then:*

- *for all $y \in \mathsf{FV}(\mathsf{gc}(rv))$ $\mathsf{type}(y, \mathcal{D}'_{qx}) = q'\ C$ where $q' = \texttt{caps}$ or $q' = \texttt{imm}$ and*
- *if $q = \texttt{imm}$ and $\mathcal{D}'_{qx}[rv] \longrightarrow^{\star} \mathcal{D}''_{qx}[rv']$ then $rv = rv'$.*

We now turn to the property of lent references, i.e., if an expression e refers to a portion of memory only through `lent` references, then the evaluation of e cannot introduce sharing between such portion of memory and external references.

To express this theorem we consider contexts, $\mathcal{E}_x$, in which the declarations preceding the hole are all evaluated. So they represent the store for the expression in the hole.

$$\mathcal{E}_x ::= \{dvs\ \ T\,x\texttt{=}[\ ]\texttt{;}\ \ ds\ e\} \mid \{dvs\ \ T\,y\texttt{=}\mathcal{E}_x\texttt{;}\ \ ds\ e\} \mid \{dvs\ \mathcal{E}_x\} \mid \mathcal{E}_x\texttt{.}f \\ \mid \mathcal{E}_x\texttt{.}f\texttt{=}e \mid v\texttt{.}f\texttt{=}\mathcal{E}_x \mid \texttt{new}\ C\texttt{(}vs\mathcal{E}_x es\texttt{)} \mid \mathcal{E}_x\texttt{.}m\texttt{(}es\texttt{)} \mid e\texttt{.}m\texttt{(}vs\mathcal{E}_x es\texttt{)}$$

The *store associate to* $\mathcal{E}_x$, dubbed $\mathsf{store}(\mathcal{E}_x)$, is :

- $\mathsf{store}(\{dvs\ \ T\,x\texttt{=}[\ ]\texttt{;}\ \ ds\ e\}) = dvs$,
- $\mathsf{store}(\{dvs\ \mathcal{E}_x\}) = \mathsf{store}(\{dvs\ \ T\,y\texttt{=}\mathcal{E}_x\texttt{;}\ \ ds\ e\}) = dvs\ \mathsf{store}(\mathcal{E}_x)$ and
- $\mathsf{store}(\mathcal{E}_x\texttt{.}f) = \cdots = \mathsf{store}(e\texttt{.}m\texttt{(}vs\mathcal{E}_x es\texttt{)}) = \mathsf{store}(\mathcal{E}_x)$

Given a store dvs we can define the *sharing relation induced by the store*, dubbed $\mathsf{Sh}(dvs)$, by considering the connections due to the rv associate to the declared (mutable) variables, as follows:

$$\mathsf{Sh}(q_1^{\tau_1} C_1\, x_1\texttt{=}rv_1\texttt{;} \cdots q_n^{\tau_n} C_n\, x_n\texttt{=}rv_n\texttt{;}) = \textstyle\sum_{1 \leq i \leq n\ \wedge\ q_i \neq \texttt{imm}} \{x_i\} \cup \mathsf{FV}(rv_i).$$

In the following $dvs(x) = dv$ is $dv = T\,x$=rv; $\in dvs$ for some T and rv. Moreover, the set of variables declared in the store associated to $\mathcal{E}_x$ is denoted by $\mathsf{Dcl}(\mathcal{E}_x)$. We can now state the property of lent references as follows.

Theorem 5 (Lent). *Let* $\vdash \mathcal{E}_x[e]$ *and for all* $y \in \mathsf{FV}(e)$ *we have* $\mathsf{store}(\mathcal{E}_x)(y) = q^{\mathtt{lent}}C\ y$=$rv$;, *for some* q *and* C. *If* $\mathcal{E}_x[e] \longrightarrow^\star \mathcal{E}'_x[e']$, *then*

$$\mathsf{Sh}(\mathsf{store}(\mathcal{E}'_x))\backslash(\mathsf{Dcl}(\mathcal{E}'_x) - \mathsf{Dcl}(\mathcal{E}_x)) \sqsubseteq \mathsf{Sh}(\mathsf{store}(\mathcal{E}_x)).$$

Consider the following simple examples of use of a lent reference. Let

$$\mathcal{E}_x = \{\mathtt{mut}\,y\mathtt{=new}\ C\mathtt{(}y\mathtt{);}\ \mathtt{mut}^{\mathtt{lent}}C\,z\mathtt{=new}\ C\mathtt{(}y\mathtt{);}\ T\,x\mathtt{=[\,];}\ e\}$$

and e_1 be $z.f$=z. We can show that $\vdash \mathcal{E}_x[e_1]$ and $\mathcal{E}_x[e_1] \longrightarrow \mathcal{E}'_x[z]$ where

$$\mathcal{E}'_x = \{\mathtt{mut}\,y\mathtt{=new}\ C\mathtt{(}y\mathtt{);}\ \mathtt{mut}^{\mathtt{lent}}C\,z\mathtt{=new}\ C\mathtt{(}z\mathtt{);}\ T\,x\mathtt{=[\,];}\ e\}.$$

Since $\mathsf{Sh}(\mathsf{store}(\mathcal{E}_x)) = \{z, y\}$ and $\mathsf{Sh}(\mathsf{store}(\mathcal{E}'_x)) = \{z\}\{y\}$ (with our notation we could use ϵ instead of $\{z\}\{y\}$) we have that $\mathsf{store}(\mathcal{E}'_x) \sqsubseteq \mathsf{store}(\mathcal{E}_x)$.
Let e_2 be $z.f$=y. We have that $\nvdash \mathcal{E}_x[e_2]$ and $\nvdash \mathcal{E}'_x[e_2]$. Indeed, letting $\Gamma = y : \mathtt{mut}\,C, z : \mathtt{mut}^{\mathtt{lent}}C$, $\Gamma \vdash z : \mathtt{mut}^{\mathtt{lent}}C \,|\, \{z, \mathtt{res}\}$ and $\Gamma \vdash y : \mathtt{mut}\,C \,|\, \{y, \mathtt{res}\}$. However, to type $z.f$=y rule (T-Field-Assign) requires $\{z, \mathtt{res}\} \subseteq \{y, \mathtt{res}\}$, which is false.

5 Conclusion and further work

We have presented a type system which combines *tracing* sharing effects possibly introduced by the evaluation of an expression with *preventing* sharing and mutation by type qualifiers which forbid some actions. Sharing is directly represented at the syntactic level as a relation among free variables, thanks to the fact that in the underlying calculus memory is encoded in terms. As shown by the examples of Sect.2, this type system is very powerful. Notably, it discriminates between well-typed and ill-typed terms in situations where type systems only based on declaring qualifiers are either too restrictive or require rather tricky rules [18,17,29].

This paper extends recent work on inference of sharing effects, see [13,15], to include `lent` constraints. In [13] we proved soundness of the type system, and theorems expressing that references declared to be capsule have the expected behaviour. Here we are adapting the theorems and extending them to the immutable qualifier. We also stated theorems saying that the type system prevents expressions using `lent` references from introducing new connections for those references. In this way expressions referring to a portion of memory only through `lent` references cannot introduce sharing between such portion of memory and external references. In a forthcoming extended version of the paper we are planning to
- provide a bidirectional type system, [12], that would allow us to to infer the sharing produced by the execution of an expression given the sharing of its context and
- give an operational semantics in which sharing is explicitly represented.
This should allow us to make more direct statements and proofs of the results, in particular for the ones for `lent` references.

Acknowledgments We would like to thank the anonymous referees for their helpful comments.

References

1. Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP'97 - Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, 1997.
2. Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journ. of Functional Programming*, 7(3):265–301, 1997.
3. Joshua Bloch. *Effective Java (2Nd Edition) (The Java Series)*. Prentice Hall PTR, 2 edition, 2008.
4. John Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exper.*, 31(6):533–553, May 2001.
5. Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2015.
6. Andrea Capriccioli, Marco Servetto, and Elena Zucca. An imperative pure calculus. *Electronic Notes in Theoretical Computer Science*, 322:87–102, 2016.
7. Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Tobias Wrigstad, and Albert Mingkun Yang. Attached and detached closures in actors. In Joeri De Koster, Federico Bergenti, and Juliana Franco, editors, *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2018, Boston, MA, USA, November 5, 2018*, pages 54–61. ACM, 2018.
8. Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013.
9. 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.
10. Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela, editors, *International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015*, pages 1–12. ACM Press, 2015.
11. Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP'07 - Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer, 2007.
12. Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 429–442. ACM, 2013.
13. Paola Giannini, Tim Richter, Marco Servetto, and Elena Zucca. Tracing sharing in an imperative pure calculus. *Science of Computer Programming*, 172:180–202, 2019.

14. Paola Giannini, Marco Servetto, and Elena Zucca. Types for immutability and aliasing control. In *ICTCS'16 - Italian Conf. on Theoretical Computer Science*, volume 1720 of *CEUR Workshop Proceedings*, pages 62–74. CEUR-WS.org, 2016.
15. Paola Giannini, Marco Servetto, and Elena Zucca. A type and effect system for uniqueness and immutability. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *SAC'18 - ACM Symp. on Applied Computing*, pages 1038–1045. ACM Press, 2018.
16. Paola Giannini, Marco Servetto, and Elena Zucca. Tracing and preventing sharing and mutation (Extended Version). Report TR-INF-2019-07-03-UNIPMN, Università Piemonte Orientale, Computer Science Institute, 2019.
17. Paola Giannini, Marco Servetto, Elena Zucca, and James Cone. Flexible recovery of uniqueness and immutability. *Theoretical Computer Science*, 764:145–172, 2019.
18. Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2012)*, pages 21–40. ACM Press, 2012.
19. John Hogg. Islands: Aliasing protection in object-oriented languages. In Andreas Paepcke, editor, *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991*, pages 271–285. ACM Press, 1991.
20. 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.
21. Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018.
22. John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journ. of Functional Programming*, 8(3):275–317, 1998.
23. Robin Milner. *Communicating and mobile systems - the Pi-calculus.* Cambridge University Press, 1999.
24. Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2007)*, pages 461–478. ACM Press, 2007.
25. Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *ACM Symp. on Principles of Programming Languages 2012*, pages 557–570. ACM Press, 2012.
26. Marco Servetto and Lindsay Groves. True small-step reduction for imperative object-oriented languages. In Werner Dietl, editor, *FTfJP'13- Formal Techniques for Java-like Programs*, pages 1:1–1:7. ACM Press, 2013.
27. 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.
28. Marco Servetto and Elena Zucca. Aliasing control in an imperative pure calculus. In Xinyu Feng and Sungwoo Park, editors, *Programming Languages and Systems - 13th Asian Symposium (APLAS)*, volume 9458 of *Lecture Notes in Computer Science*, pages 208–228. Springer, 2015.
29. Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic Java. In *ACM SIGPLAN Conference on Object-Oriented Programming,Systems, Languages and Applications (OOPSLA 2010)*, pages 598–617, 2010.