DiSIT, Computer Science Institute
Università del Piemonte Orientale "A. Avogadro"
Viale Teresa Michel 11, 15121 Alessandria
http://www.di.unipmn.it

UNIVERSITÀ DEL PIEMONTE ORIENTALE

Tracing sharing in an imperative pure calculus

P. Giannini, M. Servetto, E. Zucca (giannini@di.unipmn.it, marco.servetto@ecs.vuw.ac.nz, elena.zucca@unige.it)

TECHNICAL REPORT TR-INF-2016-12-04-UNIPMN
(December 2016)

Recent Titles from the TR-INF-UNIPMN Technical Report Series

Tracing sharing in an imperative pure calculus

Abstract

We introduce a type and effect system, for an imperative object calculus, which infers *sharing* possibly introduced by the evaluation of an expression. Sharing is directly represented at the syntactic level as a relation among free variables, thanks to the fact that the calculus is *pure*. That is, imperative features are modeled by just rewriting source code terms. We consider both standard variables and *affine* variables, which can occur at most once in their scope. The latter are used as temporary references, to "move" a *capsule* (an isolated portion of store) to another location in the store. The sharing effects inferred by the type system are very expressive, and generalize notions introduced in literature by type modifiers.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

In the imperative programming paradigm, *sharing* is the situation when a portion of the store can be accessed through more than one reference, say x and y, so that a change to x affects y as well. Unwanted sharing relations are common bugs: unless sharing is carefully maintained, changes through a reference might propagate unexpectedly, objects may be observed in an inconsistent state, and conflicting constraints on shared data may inadvertently invalidate invariants. Preventing such errors is even more important in increasingly ubiquitous multi-core and many-core architectures. An unfortunate scheduling of two threads sharing mutable state could cause data-races, which leads to problems like lost updates, corrupted data and unwanted non-determinism.

For this reasons, the last few decades have seen considerable interest in type systems for controlling sharing and interference, notably using type modifiers to restrict the usage of references. In particular, we can find in the literature variations of two notions: *lent* references [24, 12] (also called *borrowed* [19]), whose reachable graph can be manipulated, but not shared, by a client, and *capsule* references (also called *externally unique* [8], *balloon* [1, 23], *island* [14, 11], *isolated* [13])), whose reachable graph should be an isolated portion of store.

The type and effect system proposed in this paper takes a different approach. That is, rather than *declaring* the permitted sharing by type modifiers, we *infer* sharing possibly introduced by the evaluation of an expression. That is, given an expression e with free variables, e.g., x, y, z, the type system computes the following effects of the evaluation of e:

- A *sharing* relation $\mathcal{S}$ which is an equivalence relation on free variables, corresponding to the possibly introduced sharing. For instance, the expression $x\texttt{.}f\texttt{=}y\texttt{;}z\texttt{.}f$ introduces sharing only between x and y.
- A subset X of free variables, corresponding to those which will be possibly connected with (an implicit variable denoting) the result of e. For instance, the result of the expression above will be only connected to z.

In this way, the *capsule* notion becomes just a special case, that is, it is an expression whose result will be disjoint from any free variable ($X = \emptyset$). For instance, the expression $x\texttt{.}f\texttt{=}y\texttt{;new }C\texttt{(new }D\texttt{())}\texttt{.}f$ is a capsule, whereas the previous expression is not. The *lent* notion also becomes a special case: a variable x is used as lent in an expression if the evaluation of the expression will neither connect x to any other variable, nor to the result (x is a singleton in $\mathcal{S}$, and $x \notin X$). For instance x is lent in $x\texttt{.}f_1\texttt{=}x\texttt{.}f_2\texttt{;}z\texttt{.}f$. In other words, our type system generalizes lent variables (singletons) to arbitrary sets of variables.

e	$::=$	$x \mid e.f \mid e.m(es) \mid e.f{=}e' \mid \texttt{new}\ C(es) \mid \{ds\ e\}$	expression
d	$::=$	$T\,x{=}e;$	declaration
T	$::=$	C^{μ}	declaration type
μ	$::=$	$\epsilon \mid \texttt{a}$	optional modifier

Figure 1 Syntax

As the reader may have noticed, we are able to express sharing in such a natural and simple way thanks to the fact that references in the store (i.e., addresses, locations, unique object identifiers) are directly represented at the syntactic level as *variables*. Indeed, we define our type system on top of a *pure* calculus, where imperative features are modeled by just rewriting source code terms, rather than by modifying an auxiliary structure which mimics physical memory. This operational semantics will be informally introduced in Sect.2, and formalized in Sect.5.

However, not *all* the variables of our calculus correspond to references in the store. We also consider *affine* variables, which can occur at most once in their scope. These variables are used as *temporary* references, to "move" a capsule (an isolated portion of store) to another location in the store. Hence, they do not introduce sharing, and are simply ignored in the sharing relation $\mathcal{S}$ and the effects X .

The rest of the paper is organized as follows: in Sect.2 we provide syntax and an informal execution model, in Sect.3 the type system, and in Sect.4 some examples. The operational semantics of the calculus is presented in Sect.5, and the main results and proofs in Sect.6. Finally Sect.7 and Sect.8 discuss related and further work.

2 Language

The syntax of the language is given in Fig.1. 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 calculus is designed with an object-oriented flavour, inspired by Featherweight Java [15]. 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.

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 or 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.

A declaration type is a class name with an optional modifier `a`, which, if present, indicates that the variable is *affine*. An affine variable can occur at most once in its scope, and should be initialized with a *capsule*, that is, an isolated portion of store. In this way, it can be used as temporary reference, to "move" a capsule to another location in the store, without introducing sharing.

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

Fig.2 shows an example of reduction sequence in the calculus, where we emphasize at

```
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
```

Figure 2 Example of reduction

each step the redex which is reduced.

The main idea is to use variable declarations to directly represent the store. That is, a declared (non affine) variable is not replaced by its value, as in standard `let`, but the association is kept and used when necessary, as it happens, with different aims and technical problems, in cyclic lambda calculi [2, 17].

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 in Fig.2 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 fields contain `0` and `1`, respectively.

As usual, references in the store can be mutually recursive[1], 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, store is flat, as it usually happens in models of imperative languages. However, in our calculus, we are also able to represent a hierarchical store, 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. The advantage of this representation is that it models in a simple and natural way constraints about sharing among objects, notably:

- the fact that an object is not referenced from outside some enclosing object is directly modeled by the block construct: for instance, the object denoted by `y` can only be reached through `w`

[1] However, mutual recursion is not allowed between declarations which are *not* evaluated, e.g., `B x= new B(y.f); B y= new B(x.f); y` is ill-formed.

- 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`.

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
```

under a naive reduction strategy would reduce to the ill-formed term

```
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 an equivalent term where the inner block has been flattened, and get the following correct reduction sequence:

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

Formally, as in π-calculus [18], our operational semantics is defined by a *congruence* relation $\cong$, which captures structural equivalence, in addition to the reduction relation which models actual computation. Note also that in the final term the declaration of `x` can be removed (again by congruence), since useless.

Moving declaration from a block to the directly enclosing block is not always safe. For instance, in the following variant of the previous example

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

the affine variable is required to be initialized with a capsule, and this is the case indeed, since the right-hand side of the declaration is a closed block. However, by flattening the term:

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

this property would be lost, and we would get an ill-typed term. Indeed, these two terms are *not* considered equivalent in our operational model. Technically, this is obtained by detecting, during typechecking, which local variables will be connected to the result of the block, as `z` in the example, and preventing to move such declarations from a block which is the initialization expression of an affine variable.

In this case, reduction proceeds by replacing the (unique) occurrence of the affine variable by its initialization expression, as shown below.

```
Cᵃ y= {D z= new D(0); C x= new D(z,z) x} y.f ⟶
{D z= new D(0); C x= new D(z,z) x}.f ≅
D z= new D(0); C x= new D(z,z) x.f ⟶
D z= new D(0); C x= new D(z,z) z ≅
D z= new D(0); z
```

3 Type system

In this section we introduce the type and effect system for the language.

We use X, Y to range over sets of variables.

A *sharing relation* $\mathcal{S}$ is an equivalence relation on variables. 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 reachable graphs in the store are possibly shared (that is, not disjoint), hence a modification of the reachable graph of x could affect y as well, and conversely.

We syntactically represent a sharing relation $\mathcal{S}$ by a sequence[2] of sets of variables, say, $X_1 \ldots X_n$, meaning that $\mathcal{S}$ is the smallest equivalence relation which contains all pairs $\langle x, y\rangle$ which belong to some X_i. So, ϵ represents the identity relation. We define the following operations on sharing relations:

- $\mathcal{S}_1 + \mathcal{S}_2$ is the smallest equivalence relation containing $\mathcal{S}_1$ and $\mathcal{S}_2$. We also write $\mathcal{S} + Y$, meaning that Y is a sequence of one element, hence representing the sharing relation consisting of the connections $\langle x, y\rangle$ such that either $x = y$ or $x \in Y$, $y \in Y$.
- $\mathcal{S}[Y/x]$ is the sharing relation obtained by "replacing" x by Y in $\mathcal{S}$, that is, the smallest equivalence relation containing the following connections:

  $\langle y, z\rangle$, for each $\langle y, z\rangle \in \mathcal{S}$, $y \neq x, z \neq x$
  $\langle y, z\rangle$, for each $\langle x, z\rangle \in \mathcal{S}$, $y \in Y$

  We also write $X[Y/x]$, interpreting X as above.
- $\mathcal{S}\backslash Y$ is the sharing relation obtained by "removing" Y from $\mathcal{S}$, that is, the smallest equivalence relation containing the connections $\langle x, y\rangle$, for all $\langle x, y\rangle \in \mathcal{S}$, $x, y \notin Y$.
- $[Y]_{\mathcal{S}}$ is the equivalence class of Y in $\mathcal{S}$, that is, $\{x \mid \langle x, y\rangle \in \mathcal{S} \ \wedge\ y \in Y\}$.
- We say that $\mathcal{S}$ is *finer than* $\mathcal{S}'$, $\mathcal{S} \sqsubseteq \mathcal{S}'$ if for all x, $[\{x\}]_{\mathcal{S}} \subseteq [\{x\}]_{\mathcal{S}'}$. So $\mathcal{S}_i \sqsubseteq \mathcal{S}_1 + \mathcal{S}_2$ $(i = 1, 2)$.

The class table is abstractly modeled by the following functions:

- $\mathsf{fields}(C)$ gives, for each declared class C, the sequence $C_1\ f_1 \ldots C_n\ f_n$ of its fields declarations
- $\mathsf{meth}(C, m)$ gives, for each method m declared in class C, the tuple $\langle D|X|\mathcal{S}, \mu, T_1\ x_1 \ldots T_n\ x_n, e\rangle$ consisting of its return type, optional `a` modifier for `this`, parameters, and body. The return type is in turn a triple consisting of a class name D, a set of variables X, and a sharing relation $\mathcal{S}$.

The typing judgement has shape

$\Gamma \vdash e : C \mid X \mid \mathcal{S} \leadsto e'$

where Γ is a *type context*, that is, an assignment of types to variables, written $x_1{:}T_1, \ldots, x_1{:}T_n$, X is a set of variables, $\mathcal{S}$ is a sharing relation, and e' is an *annotated expression.*

The intuitive meaning is that X is the set of free variables of e which, after the evaluation of the expression, will be possibly connected with (an implicit variable denoting) the result of the expression, and $\mathcal{S}$ represents the connections among the free variables of e possibly introduced by the evaluation. Hence, the set X is always closed under $\mathcal{S}$. An expression where X is empty denotes a *capsule*, that is, reduces to an isolated portion of store. An affine

[2] In rules and most examples sequences have length one, for a sequence of length two see page 13.

$$\textsc{(t-var)}\ \frac{}{\Gamma\vdash x : C \mid \{x\} \mid \epsilon \leadsto x}\ \Gamma(x) = C \qquad \textsc{(t-affine-var)}\ \frac{}{\Gamma\vdash x : C \mid \emptyset \mid \epsilon \leadsto x}\ \Gamma(x) = C^{\mathsf{a}}$$

$$\textsc{(t-field-access)}\ \frac{\Gamma\vdash e : C \mid X \mid \mathcal{S} \leadsto e'}{\Gamma\vdash e\texttt{.}f_i : C_i \mid X \mid \mathcal{S} \leadsto e'\texttt{.}f_i}\ \begin{array}{l}\mathsf{fields}(C) = C_1\ f_1 \ldots C_n\ f_n\\ i\in 1..n\end{array}$$

$$\textsc{(t-invk)}\ \frac{\Gamma\vdash e_i : C_i \mid X_i \mid \mathcal{S}_i \leadsto e'_i \quad 0\leq i\leq n}{\begin{array}{c}\Gamma\vdash e_0\texttt{.}m\texttt{(}e_1,\ldots,e_n\texttt{)} : C \mid X \mid \mathcal{S} \leadsto\\ e'_0\texttt{.}m\texttt{(}e'_1,\ldots,e'_n\texttt{)}\end{array}}\ \begin{array}{l}\mathsf{meth}(C_0,m){=}\langle C|X'|\mathcal{S}',\mu_0, T_1\,x_1\ldots T_n\,x_n, e\rangle\\ T_i = C_i^{\mu_i}\ \ 1{\leq}i{\leq}n\\ \mu_i = \mathsf{a} \Rightarrow X_i = \emptyset \quad 0{\leq}i{\leq}n\\ X = X'[X_0/\texttt{this}][X_1/x_1]\ldots[X_n/x_n]\\ \mathcal{S}{=}\sum_{i=0}^{n}\mathcal{S}_i + \mathcal{S}'[X_0/\texttt{this}][X_1/x_1]\ldots[X_n/x_n]\end{array}$$

$$\textsc{(t-field-assign)}\ \frac{\Gamma\vdash e_1 : C \mid X_1 \mid \mathcal{S}_1 \leadsto e'_1 \qquad \Gamma\vdash e_2 : C_i \mid X_2 \mid \mathcal{S}_2 \leadsto e'_2}{\Gamma\vdash e_1\texttt{.}f_i\texttt{=}e_2 : C_i \mid [X_1\cup X_2]_{\mathcal{S}} \mid \mathcal{S} \leadsto e'_1\texttt{.}f_i\texttt{=}e'_2}\ \begin{array}{l}\mathsf{fields}(C) = C_1\ f_1 \ldots C_n\ f_n\\ i\in 1..n\\ \mathcal{S} = \mathcal{S}_1 + \mathcal{S}_2 + (X_1\cup X_2)\end{array}$$

$$\textsc{(t-new)}\ \frac{\Gamma\vdash e_i : C_i \mid X_i \mid \mathcal{S}_i \leadsto e'_i \qquad 1{\leq}i{\leq}n}{\Gamma\vdash \texttt{new}\ C\texttt{(}e_1,\ldots,e_n\texttt{)} : C \mid [\bigcup_{1\leq i\leq n} X_i]_{\mathcal{S}} \mid \mathcal{S} \leadsto \texttt{new}\ C\texttt{(}e'_1,\ldots,e'_n\texttt{)}}\ \begin{array}{l}\mathsf{fields}(C){=}C_1\ f_1 \ldots C_n\ f_n\\ \mathcal{S} = \sum_{i=1}^{n}\mathcal{S}_i\end{array}$$

$$\textsc{(t-block)}\ \frac{\begin{array}{c}\Gamma[\Gamma']\vdash e_i : C_i \mid X_i \mid \mathcal{S}_i \leadsto e'_i \qquad 1{\leq}i{\leq}n\\ \Gamma[\Gamma']\vdash e : C \mid X' \mid \mathcal{S}' \leadsto e'\end{array}}{\begin{array}{c}\Gamma\vdash \texttt{\{}T_1\ x_1\texttt{=}e_1\texttt{;}\ldots T_n\ x_n\texttt{=}e_n\texttt{;}\ e\texttt{\}} : C \mid [X']_{\mathcal{S}}\setminus Y \mid \mathcal{S}\setminus Y \leadsto\\ \texttt{\{}^{[X']_{\mathcal{S}}\cap Y}\,T_1\ x_1\texttt{=}e'_1\texttt{;}\ldots T_n\ x_n\texttt{=}e'_n\texttt{;}\ e'\texttt{\}}\end{array}}\ \begin{array}{l}\Gamma' = x_1{:}T_1,\ldots,x_n{:}T_n\\ Y = \mathsf{dom}(\Gamma')\\ T_i = C_i^{\mu_i}\ \ 1{\leq}i{\leq}n\\ \mu_i{=}\mathsf{a} \Longrightarrow\\ \quad (X_i = \emptyset \wedge x_i\ \mathsf{affine})\ \ 1{\leq}i{\leq}n\\ \mathcal{S} = \sum_{i=1}^{n}(\mathcal{S}_i + \{x_i\}{\cup}X_i) + \mathcal{S}'\end{array}$$

Figure 3 Typing rules

variable will be never connected to another, nor to the result, since it is initialized with a capsule and used only once.

Moreover, during typechecking expressions are annotated. The syntax of *annotated expressions* is given by:

$$e \quad ::= \quad x \mid e\texttt{.}f \mid e\texttt{.}m\texttt{(}e_1,\ldots,e_n\texttt{)} \mid e\texttt{.}f\texttt{=}e' \mid \texttt{new}\ C\texttt{(}es\texttt{)} \mid \texttt{\{}^{X}\mathit{ds}\ e\texttt{\}}$$

where we use the same metavariable of source expressions for simplicity. As we can see, blocks are annotated by the set X of the local variables which will be (possibly) connected with the result of the body. Such annotations, as we will see in the next section, are used to correctly define the congruence relation among terms. In particular, they prevent from moving outside of a block which initializes an affine variable declarations of variables which will be possibly connected to the result. Indeed, moving such declarations would make the term ill-typed, as shown in the last example of Sect.2.

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

- $\Gamma\vdash e : D \mid X \mid \mathcal{S} \leadsto e'$, with
- $\Gamma = \texttt{this}{:}C^{\mu}, x_1{:}T_1,\ldots,x_n{:}T_n$.

The typing rules are given in Fig.3.

In rule (t-var), the evaluation of a (non affine) variable does not introduce any connection. So the resulting sharing relation is the identity. Moreover, the result is connected only with the variable itself. In rule (T-AFFINE-VAR), the evaluation of an affine variable does not introduce any connection, and the variable is not even connected to the result. Indeed, affine variables are temporary references and will be substituted with capsules.

In rule (T-FIELD-ACCESS), the connections introduced by a field access are those introduced by the evaluation of the receiver.

In rule (T-INVK), if a parameter (including the implicit `this` parameter) is affine, then the corresponding argument is required to be a capsule, that is, its result should have no connections. The connections among free variables introduced by a method call are those introduced by the evaluation of the receiver and the arguments ($\sum_{i=0}^{n} \mathcal{S}_i$), plus those introduced by the evaluation of the method body ($\mathcal{S}'$) where `this` and parameters have been replaced by the (equivalence classes of) the receiver and the arguments, respectively. Analogously, the connections with the result are those of the method body, with the same substitution. For instance, if method m has parameters x and y, and the evaluation of its body connects x with `this`, and y with the result, then, given the method call $z\texttt{.}m\texttt{(}x', y'\texttt{)}$, its evaluation connects x' with z, and y' with the result.

In rule (T-FIELD-ASSIGN), the connections among free variables introduced by a field assignment are those introduced by the evaluation of the two sides ($\mathcal{S}_1$ and $\mathcal{S}_2$), plus the connection between the (equivalence classes of) the two results. The connections with the result are the variables in the equivalence classes of those of the two sides in the resulting sharing relation. For instance, given the assignment $e\texttt{.}f\texttt{=}e'$, if the evaluation of e connects y with z and x with the result, and the evaluation of e' connects y' with z' and x' with the result, then the evaluation of the field assignment connects y with z, y' with z', and connects together x, x' and the result.

In rule (T-NEW), the connections among free variables introduced by a constructor invocation are those introduced by the evaluation of the arguments. The connections with the result are the variables in the equivalence classes of those of the arguments of the constructor.

In rule (T-BLOCK), the initialization expressions and the body of the block are typechecked in the current type context, enriched by the association to local variables of their declaration types. We denote by $\Gamma[\Gamma']$ the type environment which is equal to Γ' when Γ' is defined, to Γ otherwise. If a local variable is affine, then its initialization expression is required to denote a capsule. Moreover, the variable can occur at most once in its scope, as abbreviated by the side condition "x_i affine".[3] The connections among free variables are obtained by:

- collecting those introduced by the evaluation of the initialization expressions ($\sum_{i=1}^{n} \mathcal{S}_i$) and the body ($\mathcal{S}'$), plus, for each declared variable, the connection between the (equivalence class of) the result and the variable itself, represented by the (single) equivalence class $\{x_i\} \cup X_i$;
- then, removing from the resulting sharing relation $\mathcal{S}$ the local variables.

The connections with the result are the variables in the equivalence class of those of the body in $\mathcal{S}$, again removing the local ones. The block is annotated with the subset of local variables which are in the sharing relation $\mathcal{S}$ with the variables connected to the result.

For a sequence of declarations ds, and a set of variables X, we define the subsequence of ds containing the declarations of variables that are connected to variables in X.

[3] In our case the affinity requirement can be simply expressed as syntactic well-formedness condition, rather than by context rules, as in linear logic-style type systems.

▶ **Definition 1.** Given X, and ds, define $ds_{|X}$, by $T_x\ x\texttt{=}e_x\texttt{;} \in ds_{|X}$ if $T_x\ x\texttt{=}e_x\texttt{;} \in ds$, and either $x \in X$ or $x \in \mathsf{FV}(e_z)$ for some $T_z\ z\texttt{=}e_z\texttt{;} \in ds_{|X}$.

The following proposition formalizes the relevant properties of the type judgement, some informally mentioned above.

▶ **Proposition 1.** Let $\Gamma \vdash e : C \mid X \mid \mathcal{S} \rightsquigarrow e'$. Then

1. if $x \in X$, then $\Gamma(x) = C$
2. if $\langle x, y\rangle \in \mathcal{S}$, $x \neq y$, then $\Gamma(x) = \Gamma(y) = C$
3. X is an equivalence class of $\mathcal{S}$
4. if $e = \texttt{\{}ds\ e_b\texttt{\}}$, then $e' = \texttt{\{}^{Y} ds'\ e'_b\texttt{\}}$, for some e'_b, ds', and Y such that $Y \subseteq \mathsf{dom}(ds_{|\mathsf{FV}(e_b)})$

Proof. By induction on type derivations. ◀

4 Examples

In this section we illustrate the expressiveness of the type system by programming examples, and for some of them we show the type derivation. In particular, we show that significant examples from [24, 12] are typechecked here in a much simpler and natural way.

▶ **Example 2.** Assume we have a class `D` with a field of type `D`, and a class `C` with two fields of type `C`. Consider the following closed expression `e`:

```
D y= new D(y);
D x= new D(x);
C^a z= {D z2= new D(z2); D z1= (y.f= x); new C(z2,z2)};
z
```

The inner block (right-hand side of the declaration of `z`) refers to the external variables `x` and `y`, that is, they occur free in the block. In particular, the execution of the block has the sharing effect of connecting `x` and `y`. However, such variables will *not* be connected to the final result of the block, since the result of the assignment will be only connected to a local variable which is not used to build the final result, as more clearly shown by using the sequence abbreviation: `{D z2= new D(z2); y.f= x; new C(z2,z2)}`.

Indeed, as will be shown in next section, the block reduces to
`{D z2= new D(z2); new C(z2,z2)}` which is a closed block.

In existing type systems supporting the capsule notion this example is either ill-typed [13], or can be typed by means of a rather tricky *swap* typing rule [24, 12] which, roughly speaking, temporarily changes, in a subterm, the set of variables which can be freely used.

▶ **Example 3.** As a counterexample, consider the following ill-typed term

```
D y= new D(y);
D x= new D(x);
C^a z= {D z1= (y.f= x); new C(z1,z1)};
z
```

Here the inner block is not a capsule, since the local variable `z1` is initialized as an *alias* of `x`, hence the final result will be connected to both `x` and `y`. Formally, the block reduces to `new C(y,y)` which is not closed.

Type derivation for Example 2

Let $\Gamma_1 = \mathtt{y}{:}\mathtt{D}, \mathtt{x}{:}\mathtt{D}, \mathtt{z}{:}\mathtt{C}^{\mathtt{a}}$, and $\Gamma_2 = \mathtt{z2}{:}\mathtt{D}, \mathtt{z1}{:}\mathtt{D}$.

In Fig.4 we give the type derivation that shows that expression `e` of Example 2 is well-typed. To save space we omit the annotated expression produced by the derivation, and show the annotations for the blocks at the bottom of the figure.

The type derivation $\mathcal{D}_2$ produces the judgement $\Gamma_1 \vdash \mathtt{e1} : \mathtt{C} \mid \emptyset \mid \{\mathtt{x}, \mathtt{y}\}$ where `e1` is the block on the right-hand-side of the declaration of `z`. The sharing relation resulting from the evaluation of the right-hand side of the declarations and the body is represented by $\{\mathtt{x}, \mathtt{y}\}$ (`z1` and `z2` are only connected with themselves). The block is annotated with $\{\mathtt{z2}\}$. Since `e1` denotes a capsule, it may be used to initialize an affine variable.
$\mathcal{D}$ is the type derivation for the whole expression `e`. Note that, since `z` is an affine variable, the body of the block (`z`) is not connected to any variable, so the annotation of the block `e` is $\emptyset$.

$$\mathcal{D}_1 : \quad \frac{\dfrac{\begin{array}{c}\Gamma_1[\Gamma_2] \vdash \mathtt{x} : \mathtt{D} \mid \{\mathtt{x}\} \mid \epsilon \\ \Gamma_1[\Gamma_2] \vdash \mathtt{y} : \mathtt{D} \mid \{\mathtt{y}\} \mid \epsilon\end{array}}{\Gamma_1[\Gamma_2] \vdash \mathtt{y.f{=}x} : \mathtt{D} \mid \{\mathtt{x}, \mathtt{y}\} \mid \{\mathtt{x}, \mathtt{y}\}}}{\Gamma_1[\Gamma_2] \vdash \mathtt{new\ D(y.f{=}x)} : \mathtt{D} \mid \{\mathtt{x}, \mathtt{y}\} \mid \{\mathtt{x}, \mathtt{y}\}}$$

$$\mathcal{D}_2 : \quad \frac{\dfrac{\Gamma_1[\Gamma_2] \vdash \mathtt{z2} : \mathtt{D} \mid \{\mathtt{z2}\} \mid \epsilon}{\Gamma_1[\Gamma_2] \vdash \mathtt{new\ D(z2)} : \mathtt{D} \mid \{\mathtt{z2}\} \mid \epsilon} \qquad \mathcal{D}_1 \qquad \dfrac{\Gamma_1[\Gamma_2] \vdash \mathtt{z2} : \mathtt{D} \mid \{\mathtt{z2}\} \mid \epsilon}{\Gamma_1[\Gamma_2] \vdash \mathtt{new\ C(z2, z2)} : \mathtt{C} \mid \{\mathtt{z2}\} \mid \epsilon}}{\Gamma_1 \vdash \mathtt{\{D\ z2{=}new\ D(z2);\ D\ z1{=}new\ D(y.f{=}x);\ new\ C(z2, z2)\}} : \mathtt{C} \mid \emptyset \mid \{\mathtt{x}, \mathtt{y}\}}$$

$$\mathcal{D} : \quad \frac{\dfrac{\Gamma_1 \vdash \mathtt{y} : \mathtt{D} \mid \{\mathtt{y}\} \mid \epsilon}{\Gamma_1 \vdash \mathtt{new\ D(y)} : \mathtt{D} \mid \{\mathtt{y}\} \mid \epsilon} \qquad \dfrac{\Gamma_1 \vdash \mathtt{x} : \mathtt{D} \mid \{\mathtt{x}\} \mid \epsilon}{\Gamma_1 \vdash \mathtt{new\ D(x)} : \mathtt{D} \mid \{\mathtt{x}\} \mid \epsilon} \qquad \mathcal{D}_2 \qquad \Gamma_1 \vdash \mathtt{z} : \mathtt{C} \mid \emptyset \mid \epsilon}{\vdash \mathtt{\{D\ y{=}new\ D(y);\ D\ x{=}new\ D(x);\ C^{a}\ z{=}e1;\ z\}} : \mathtt{C} \mid \emptyset \mid \epsilon}$$

- $\mathcal{D}_2$ yields $\mathtt{e1}' = \{^{\{\mathtt{z2}\}}\mathtt{D\ z2{=}new\ D(z2);\ D\ z1{=}new\ D(y.f{=}x);\ new\ C(z2, z2)\}}$
- $\mathcal{D}$ yields $\mathtt{e}' = \{^{\emptyset}\mathtt{D\ y{=}new\ D(y);\ D\ x{=}new\ D(x);\ C^{a}\ z{=}e1';\ z\}}$

Figure 4 Type derivation for Example 2

▶ **Example 4.** We provide now a more realistic programming example, assuming a syntax enriched by usual programming constructs. The class `CustomerReader` below models reading information about customers out of a text file formatted as shown in the example:

```
Bob
1 500 2 1300
Mark
42 8 99 100
```

In even lines we have customer names, in odd lines we have a shop history: a sequence of product codes. The method `CustomerReader.read` takes a `Scanner`, assumed to be a class similar to the one in Java, for reading a file and extracting different kinds of data.

```
class CustomerReader {
  static Customer read(Scanner s)/*$X = \emptyset$ $\mathcal{S} = \epsilon$*/{
        Customer c=new Customer(s.nextLine())
      while(s.hasNextNum()){
        c.addShopHistory(s.nextNum())
      }
      return c
  }
}

class Scanner {
  String nextLine()/*$X = \emptyset$ $\mathcal{S} = \epsilon$*/{...}
  boolean hasNextNum()/*$X = \emptyset$ $\mathcal{S} = \epsilon$*/{...}
  int nextNum()/*$X = \emptyset$ $\mathcal{S} = \epsilon$*/{...}
```

Here and in the following, we insert after method headers, as comments, their sharing effects. In a real language, a library should declare sharing effects of methods by some concrete syntax, as part of the type information available to clients. In this example, `CustomerReader.read` uses some methods of class `Scanner`. Having no parameters besides `this`, for such methods the sharing relation is necessarily the identity, represented by $\mathcal{S} = \epsilon$. Moreover, their result is not connected to the receiver, as specified by $X = \emptyset$ (for the last two methods this is necessarily the case since the result is a primitive value).

A `Customer` object is read from the file, and then its shop history is added. Since methods invoked on the scanner introduce no sharing, we can infer that the same holds for method `CustomerReader.read`. In other words, we can statically ensure that the data of the scanner are not mixed with the result. In previous work [24, 12] the same guarantee was obtained by declaring `lent` (borrowed) the `Scanner s` parameter. In our type system, the fact that a parameter is (used as) lent, that is, it will be neither connected to another parameter, nor to the result, is *inferred* instead. Moreover, lent parameters (singletons) are generalized to arbitrary sets of parameters, as will be shown in Example 5.

The following method `update` illustrates how we can "open" capsules, modify their values and then recover the original capsule guarantee. The method takes a customer, which is required to be a capsule by the fact that the corresponding parameter is affine, and a scanner as before.

```
class CustomerReader {...//as before
  static Customer update(Customer$^a$ old,Scanner s)/*$X = \emptyset$ $\mathcal{S} = \epsilon$*/{
    Customer c=old//we open the capsule 'old'
    while(s.hasNextNum()){
      c.addShopHistory(s.nextNum())
    }
    return c
  }
}
```

Every method which only has affine and lent parameters can use the pattern illustrated above: one (or many) affine parameters are opened (that is, assigned to local variables) and, in the end, the result is guaranteed to be a capsule again. This mechanism is not possible in [1, 8, 11] and relies on destructive reads in [13].

A less restrictive version of method `update` could take a non affine `Customer old` parameter, that is, not require that the old customer is a capsule. In this case, the sharing effects would be $X = \{\texttt{old}\}$ $\mathcal{S} = \epsilon$. Hence, in a call `Customer.update(c,s)`, the connections of `c` would be propagated to the result of the call. In other words, the method type is "polymorphic" with respect to sharing effects.

▶ **Example 5.** The following method takes two teams `t1`, `t2`. Both teams want to add a reserve player from their respective lists `p1` and `p2`, assumed to be sorted with best players first. However, to keep the game fair, the two reserve players can only be added if they have the same skill level.

```
static void addPlayer(Team t1, Team t2, Players p1, Players p2)
/*$X = \emptyset$ $\mathcal{S} = \{\texttt{t1},\texttt{p1}\},\{\texttt{t2},\texttt{p2}\}$*/{
  while(true){//could use recursion instead
    if(p1.isEmpty()||p2.isEmpty()) {/*error*/}
    if(p1.top().skill==p2.top().skill){
      t1.add(p1.top());
      t2.add(p2.top());
      return;
      }
    else{
      removeMoreSkilled(p1,p2);
      }
  }
```

The sharing effects express the fact that each team is only mixed with its list of reserve players.

▶ **Example 6.** Finally, we provide a more involved example which illustrates the expressive power of our approach. Assume we have a class `C` as follows:

```
class C {
   C f;
   C clone()/*$X = \emptyset$ $\mathcal{S} = \epsilon$*/{...}
   C mix(C x)/*$X = \{\texttt{x},\texttt{this}\}$ $\mathcal{S} = \{\texttt{x},\texttt{this}\}$*/{...}
}
```

The method `clone` is expected to return a deep copy of the receiver. Indeed, this method has no parameters, apart the implicit non affine parameter `this`, and returns an object of class `C` which is *not* connected to the receiver, as specified by the set $\emptyset$. Having no parameters besides `this`, the sharing relation is necessarily the identity, represented by ϵ. Note that a shallow clone method would be typed `C clone()/*`$X = \{\texttt{this}\}$ $\mathcal{S} = \epsilon$`*/`.

The method `mix` is expected to return a "mix" of the receiver with the argument. Indeed, this method has, besides `this`, a parameter `x` of class `C`, both non affine, returns an object of class `C` and its effects are connecting `x` with `this`, and both with the result.

Consider now the following closed expression `e`:

```
C c1= new C(c1);
C outC= {
  C c2= new C(c2);
  C$^a$ inC= {
    C c3= new C(c3);
    C res= c2.mix(c1).clone()
    res.mix(c3)};
  inC.mix(c2)};
outC
```

$$\mathcal{D}_1 : \quad \dfrac{\dfrac{\begin{array}{c}\Gamma \vdash \texttt{c1} : \texttt{C} \mid \{\texttt{c1}\} \mid \epsilon \\ \Gamma \vdash \texttt{c2} : \texttt{C} \mid \{\texttt{c2}\} \mid \epsilon\end{array}}{\Gamma \vdash \texttt{c2.mix(c1)} : \texttt{C} \mid \{\texttt{c1},\texttt{c2}\} \mid \{\texttt{c1},\texttt{c2}\}}}{\Gamma \vdash \texttt{c2.mix(c1).clone()} : \texttt{C} \mid \emptyset \mid \{\texttt{c1},\texttt{c2}\}} \qquad \mathcal{D}_2 : \quad \dfrac{\begin{array}{c}\Gamma \vdash \texttt{res} : \texttt{C} \mid \{\texttt{res}\} \mid \epsilon \\ \Gamma \vdash \texttt{c3} : \texttt{C} \mid \{\texttt{c3}\} \mid \epsilon\end{array}}{\Gamma \vdash \texttt{res.mix(c3)} : \texttt{C} \mid \{\texttt{c3},\texttt{res}\} \mid \{\texttt{c3},\texttt{res}\}}$$

$$\mathcal{D}_3 : \quad \dfrac{\dfrac{\Gamma \vdash \texttt{c3} : \texttt{C} \mid \{\texttt{c3}\} \mid \epsilon}{\Gamma \vdash \texttt{new C(c3)} : \texttt{C} \mid \{\texttt{c3}\} \mid \epsilon} \qquad \mathcal{D}_1 \qquad \mathcal{D}_2}{\Gamma_1[\Gamma_2] \vdash \texttt{\{C c3=new C(c3); C res=c2.mix(c1).clone(); res.mix(c3)\}} : \texttt{C} \mid \emptyset \mid \{\texttt{c1},\texttt{c2}\}}$$

$$\mathcal{D}_4 : \quad \dfrac{\dfrac{\Gamma_1[\Gamma_2] \vdash \texttt{c2} : \texttt{C} \mid \{\texttt{c2}\} \mid \epsilon}{\Gamma_1[\Gamma_2] \vdash \texttt{new C(c2)} : \texttt{C} \mid \{\texttt{c2}\} \mid \epsilon} \qquad \mathcal{D}_3 \qquad \dfrac{\begin{array}{c}\Gamma_1[\Gamma_2] \vdash \texttt{inC} : \texttt{C} \mid \emptyset \mid \epsilon \ (\textsc{t-var}) \\ \Gamma_1[\Gamma_2] \vdash \texttt{c2} : \texttt{C} \mid \{\texttt{c2}\} \mid \epsilon \ (\textsc{t-var})\end{array}}{\Gamma_1[\Gamma_2] \vdash \texttt{inC.mix(c2)} : \texttt{C} \mid \{\texttt{c2}\} \mid \epsilon}}{\Gamma_1 \vdash \texttt{\{C c2=new C(c2); C}^{\texttt{a}}\texttt{ inC=e1; inC.mix(c2)\}} : \texttt{C} \mid \{\texttt{c1}\} \mid \epsilon}$$

$$\mathcal{D} : \quad \dfrac{\dfrac{\Gamma_1 \vdash \texttt{c1} : \texttt{C} \mid \{\texttt{c1}\} \mid \epsilon \ (\textsc{t-var})}{\Gamma_1 \vdash \texttt{new C(c1)} : \texttt{C} \mid \{\texttt{c1}\} \mid \epsilon}(\textsc{t-new}) \qquad \mathcal{D}_4 \qquad \Gamma_1 \vdash \texttt{outC} : \texttt{C} \mid \{\texttt{outC}\} \mid \epsilon \ (\textsc{t-var})}{\vdash \texttt{\{C c1=new C(c1); C outC=e2; outC\}} : \texttt{C} \mid \emptyset \mid \epsilon}$$

- $\mathcal{D}_3$ yields $\texttt{e1}' = \{^{\{\texttt{res},\texttt{c3}\}}$`C c3=new C(c3); C res=c2.mix(c1).clone(); res.mix(c3)}`
- $\mathcal{D}_4$ yields $\texttt{e2}' = \{^{\{\texttt{c2}\}}$`C c2=new C(c2);` $\texttt{C}^{\texttt{a}}$ `inC=e1; inC.mix(c2)}`
- $\mathcal{D}$ yields $\texttt{e}' = \{^{\{\texttt{outC}\}}$`C c1=new C(c1); C outC=e2; outC}`

Figure 5 Type derivation for Example 6

The key line in this example is `C res=c2.mix(c1).clone()`.
Thanks to the fact that `clone` returns a capsule, we know that `res` will not be connected to the external variables `c1` and `c2`, hence also the result of the block will not be connected to `c1` or `c2`. However, the sharing between `c2` and `c1` introduced by the `mix` call is traced, and prevents the *outer* block from being a capsule. The reader can check that, by replacing the declaration of `res` with `C res=c2.mix(c2).clone()`, also the outer block turns out to be a capsule, hence variable `outC` could be declared affine. Existing type systems supporting the capsule notion [13, 24, 12] either do not discriminate between these two cases, or require rather tricky and non syntax-directed rules.

Note that, none of the variables, except for `inC`, could be declared with the affine modifier. Indeed the variables `c1`, `c2`, and `c3` appear more than once, and `outC`, as already said, is initialized with an expression whose result could be connected with the free variable `c1`.

Type derivation for Example 6

Let $\Gamma_1 = \texttt{c1:C}, \texttt{outC:C}$, $\Gamma_2 = \texttt{c2:C}, \texttt{inC:C}^{\texttt{a}}$, and $\Gamma_3 = \texttt{c3:C}, \texttt{res:C}$.

In Fig.5 we give the type derivation that shows that the expression `e` of Example 6 is well-typed.

Derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ end with an application of rule (T-Invk). Consider $\mathcal{D}_2$. The method `mix` produces sharing between its receiver and parameter, and also its result is connected

e	$::=$	$x \mid v.f \mid v.m(vs) \mid v.f{=}v \mid \texttt{new}\ C(vs) \mid \{ds\ e\}$	expression
d	$::=$	$T\,x{=}e;$	declaration
v	$::=$	$x \mid \{^X dvs\ v\}$	value
dv	$::=$	$C\,x{=}\texttt{new}\ C(xs);$	evaluated declaration
$\mathcal{E}$	$::=$	$[\,] \mid \{^X dvs\ T\,x{=}\mathcal{E};\ ds\ e\} \mid \{^X dvs\ \mathcal{E}\}$	evaluation context

Figure 6 Syntax of calculus, values, evaluated declarations, and evaluation contexts

with receiver and parameter, then the call of `mix` with receiver `res` and parameter `c3` returns a result connected with both these variables and produces sharing between them. The call of `mix` with receiver `c2` and parameter `c1` in the derivation $\mathcal{D}_1$ does the same with `c2` and `c1`. The type derivation $\mathcal{D}_3$ justifies the judgement $\Gamma_1[\Gamma_2] \vdash \texttt{e1} : \texttt{C} \mid \emptyset \mid \{\texttt{c1}, \texttt{c2}\}$ where `e1` is the inner block, the initialization expression of `inC`. The sharing relation resulting from the evaluation of the right-hand side of the declarations and the body is represented by $\{\texttt{c1}, \texttt{c2}\}\,\{\texttt{c3}, \texttt{res}\}$. The block denotes a capsule, since the result of the body of the block, `res.mix(c3)`, is connected only to local variables. Therefore, `e1` can be used as initialization expression of the affine variable `inC`. The annotation for the block, i.e. the set of local variables that may be connected to the result, is $\{\texttt{res}, \texttt{c3}\}$, so, when applying the congruence relation, the variables `res` and `c3` cannot be moved outside this block. The type derivation $\mathcal{D}_4$ justifies the judgement $\Gamma_1 \vdash \texttt{e2} : \texttt{C} \mid \{\texttt{c1}\} \mid \epsilon$, where `e2` is the block which is the initialization expression of `outC`. The sharing relation resulting from the evaluation of initialization expressions and body is represented by $\{\texttt{c1}, \texttt{c2}\}$ (this sharing is produced by the evaluation of `e1`). The variable `inC`, being affine, is not connected to any other variable (including `inC` itself). Indeed, it will be substituted with the result of the evaluation of `e1` and so it will disappear. Only the local variable `c2` may be connected to the result, so the block is annotated with $\{\texttt{c2}\}$. Therefore, this block is not a capsule, and could not be used to initialize an affine variable. Finally, $\mathcal{D}$ is the derivation for the expression `e`. The block is a closed expression, and closed expressions are capsules. The block is annotated with the local variables `outC`, which is its body.

5 The calculus

The calculus, defined in Fig.6, has a simplified syntax where we assume that, except from right-hand sides of declarations and body of blocks, subterms of a compound expression are only values. This simplification can be easily obtained by a (type-driven) translation of the syntax of Fig.1 generating for each subterm which is not a value a local declaration of the appropriate type.

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.

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 an *object state* $\texttt{new}\ C(xs)$ to a variable (reference). An object state represents an elementary allocation unit, and is a shorter form for a block $\{C\,x{=}\texttt{new}\ C(xs);\ x\}$, as formalized by congruence rule (NEW) in Fig.7.

As anticipated in Sect.2, we allow mutual recursion only among evaluated declarations, 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 as in [22]. Here we take a simplifying assumption since this is not the focus of this paper.

Evaluation contexts express standard left-to-right evaluation. In the evaluation context $\texttt{\{}^X \mathit{dvs}\ T\ x\texttt{=}\mathcal{E}\texttt{;}\ \mathit{ds}\ e\texttt{\}}$ we assume that no declaration in ds is evaluated. Indeed, we can always move evaluated declarations first, as formalized by congruence rule (reorder) in Fig.7.

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 [18].

Both relations are defined on terms obtained through the typechecking phase, hence where blocks have been annotated as described in the previous section. Formally, given an annotated term e, and denoting by e^- the term obtained by erasing annotations from e, we assume that $\Gamma \vdash e^- : C \mid X \mid \mathcal{S} \leadsto e$, for some Γ, C, X, and $\mathcal{S}$.

In the following, for simplicity we show annotations only when they are relevant.

The congruence relation, denoted by $\cong$, is defined as the smallest congruence satisfying the axioms in Fig.7. We write $\mathsf{FV}(\mathit{ds})$ and $\mathsf{FV}(e)$ for the free variables of a sequence of declarations and an expression, respectively, and $X[y/x]$, $\mathit{ds}[y/x]$, and $e[y/x]$ for the capture-avoiding variable substitution on a set of variables, a sequence of declarations, and an expression, respectively, all defined in the standard way.

$$\text{(alpha)}\quad \texttt{\{}^X \mathit{ds}\ T\ x\texttt{=}e\texttt{;}\ \mathit{ds}'\ e'\texttt{\}} \cong \texttt{\{}^{X[y/x]} \mathit{ds}[y/x]\ T\ y\texttt{=}e[y/x]\texttt{;}\ \mathit{ds}'[y/x]\ e'[y/x]\texttt{\}}$$

$$\text{(reorder)}\quad \texttt{\{}^X \mathit{ds}\ C\ x\texttt{=new}\ C\texttt{(}\mathit{xs}\texttt{);}\ \mathit{ds}'\ e\texttt{\}} \cong \texttt{\{}^X C\ x\texttt{=new}\ C\texttt{(}\mathit{xs}\texttt{);}\ \mathit{ds}\ \mathit{ds}'\ \ e\texttt{\}}$$

$$\text{(new)}\quad \texttt{new}\ C\texttt{(}\mathit{vs}\texttt{)} \cong \texttt{\{}^{\{x\}} C\ x\texttt{=new}\ C\texttt{(}\mathit{vs}\texttt{);}\ x\texttt{\}}$$

$$\text{(garbage)}\quad \texttt{\{}^X \mathit{dvs}\ \mathit{ds}\ e\texttt{\}} \cong \texttt{\{}^{X\setminus\mathsf{dom}(\mathit{dvs})} \mathit{ds}\ e\texttt{\}} \qquad (\mathsf{FV}(\mathit{ds}) \cup \mathsf{FV}(e)) \cap \mathsf{dom}(\mathit{dvs}) = \emptyset$$

$$\text{(block-elim)}\quad \texttt{\{}^\emptyset\ e\texttt{\}} \cong e$$

$$\text{(body)}\quad \texttt{\{}^Y \mathit{ds}\ \texttt{\{}^X \mathit{ds}_1\ \mathit{ds}_2\ e\texttt{\}\}} \cong \texttt{\{}^{Y\cup(X\cap\mathsf{dom}(\mathit{ds}_1))} \mathit{ds}\ \mathit{ds}_1\ \texttt{\{}^{X\setminus\mathsf{dom}(\mathit{ds}_1)} \mathit{ds}_2\ e\texttt{\}\}} \qquad \begin{array}{l}\mathsf{FV}(\mathit{ds}_1) \cap \mathsf{dom}(\mathit{ds}_2) = \emptyset \\ \mathsf{FV}(\mathit{ds}) \cap \mathsf{dom}(\mathit{ds}_1) = \emptyset\end{array}$$

$$\text{(dec)}\quad \begin{array}{l}\texttt{\{}^Y \mathit{ds}\ C^\mu\ x\texttt{=\{}^X \mathit{ds}_1\ \mathit{ds}_2\ e\texttt{\};}\ \mathit{ds}'\ e'\texttt{\}} \cong \\ \qquad \texttt{\{}^Y \mathit{ds}\ \mathit{ds}_1\ C^\mu\ x\texttt{=\{}^{X\setminus\mathsf{dom}(\mathit{ds}_1)} \mathit{ds}_2\ e\texttt{\};}\ \mathit{ds}'\ e'\texttt{\}}\end{array} \qquad \begin{array}{l}\mathsf{FV}(\mathit{ds}_1) \cap \mathsf{dom}(\mathit{ds}_2) = \emptyset \\ \mathsf{FV}(\mathit{ds}\ \mathit{ds}'\ e') \cap \mathsf{dom}(\mathit{ds}_1) = \emptyset \\ \mu = \mathsf{a} \implies \mathsf{dom}(\mathit{ds}_1) \cap X = \emptyset\end{array}$$

$$\text{(val-ctx)}\quad \mathcal{V}[\texttt{\{}^X \mathit{dvs}_1\ \mathit{dvs}_2\ v\texttt{\}}] \cong \texttt{\{}^{X\cap\mathsf{dom}(\mathit{dvs}_1)} \mathit{dvs}_1\ \mathcal{V}[\texttt{\{}^{X\setminus\mathsf{dom}(\mathit{dvs}_1)} \mathit{dvs}_2\ v\texttt{\}}]\texttt{\}} \qquad \begin{array}{l}\mathsf{FV}(\mathit{dvs}_1) \cap \mathsf{dom}(\mathit{dvs}_2) = \emptyset \\ \mathsf{FV}(\mathcal{V}) \cap \mathsf{dom}(\mathit{dvs}_1) = \emptyset\end{array}$$

Figure 7 Congruence rules

Rule (alpha) is the usual α-conversion. The condition $x, y \notin \mathsf{dom}(\mathit{ds}\ \mathit{ds}')$ is implicit by well-formedness of blocks.

Rule (reorder) states that we can move evaluated declarations in an arbitrary order. Note that, ds and ds' cannot be swapped in our rules, because this could change the order of side effects.

In rule (new), a constructor invocation can be seen as an elementary block where a new object is allocated.

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 they do not have side effects. Rule (block-elim) states the obvious fact that a block with no declarations is equivalent to its body.

With 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 [18].

In rules (body) and (dec), the inner block is the body, or the right-hand side of a declaration, respectively, of the enclosing block. The first two side conditions ensure that moving the declarations ds_1 does not cause either scope extrusion or capture of free variables. More precisely: the former prevents to move outside a declaration in ds_1 which depends on local variables of the inner block. The latter prevents capturing with ds_1 free variables of the enclosing block. Note that the second condition can be obtained by α-conversion of the inner block, but the first cannot. Finally, the third side condition of rule (dec) prevents, in case the block initializes an affine variable, to move outside declarations of variables that will be possibly connected to the result of the block. Indeed, in this case we would get an ill-typed term. In case of a non affine declaration, instead, this is not a problem.

Rule (valu-ctx) handles the cases when the inner block is a subterm of a field access, method invocation, field assignment or constructor invocation. Define a *value context* $\mathcal{V}$ to be any position of the previous expressions in which we could have a value, i.e.,

$$\mathcal{V} ::= [\,] \mid \mathcal{V}\texttt{.}f \mid \mathcal{V}\texttt{.}f\texttt{=}v \mid v\texttt{.}f\texttt{=}\mathcal{V} \mid \texttt{new } C\texttt{(}vs, \mathcal{V}, vs'\texttt{)}$$

In case in the position defined by the hole of $\mathcal{V}$ we have a block value, some of its (evaluated) declarations can be moved outside the construct. As for rule (body), the first side condition prevents moving outside a declaration in dvs which depends on local variables of the inner block. The following side condition prevents capturing some free variables of $\mathcal{V}$. E.g., if $\mathcal{V} = \texttt{new } C\texttt{(}vs, [\,], vs'\texttt{)}$, we would have $\mathcal{V}[\texttt{\{}^{X} dvs\ v\texttt{\}}] = \texttt{new } C\texttt{(}vs, \texttt{\{}^{X} dvs\ v\texttt{\}}, vs'\texttt{)}$, and $\texttt{\{}^{X} dvs\ \mathcal{V}[v]\texttt{\}} = \texttt{\{}^{X} dvs\ \texttt{new } C\texttt{(}vs, \texttt{\{}^{\emptyset}\ v\texttt{\}}, vs'\texttt{)}\texttt{\}}$. If some of the free variables of either vs or vs' are in the domain of dvs, they would be captured by the declaration dvs. This condition can be satisfied by α-conversion of such variables in $\texttt{\{}^{X} dvs\ v\texttt{\}}$. In this example, the first condition is trivially satisfied since we move all the declarations of the block.

The following proposition shows that values are either references or congruent to blocks whose body is a variable and that do not contain useless evaluated declarations.

▶ **Proposition** 2 (Canonical Form for Values). If v is a value, then $v \cong v'$ where $v' = x$, or $v' = \texttt{\{}^{X} dvs\ x\texttt{\}}$, where $dvs \neq \epsilon$ and $dvs_{|x} = dvs$.

Proof. Let $v = \texttt{\{}^{X} dvs\ u\texttt{\}}$. By induction on u.
<u>Case: $u = x$</u>. Using rule (reorder), we have that $\texttt{\{}^{X} dvs'\ dvs_{|x}\ x\texttt{\}} \cong \texttt{\{}^{X} dvs\ x\texttt{\}}$ for some dvs'. By definition of $dvs_{|x}$, $(\mathsf{FV}(dvs_{|x}) \cup \{x\}) \cap \mathsf{dom}(dvs') = \emptyset$. Therefore, applying rule (garbage), $\texttt{\{}^{X} dvs_{|x}\ x\texttt{\}} \cong \texttt{\{}^{X-\mathsf{dom}(dvs')} dvs\ x\texttt{\}}$. If $dvs_{|x}$ is not empty then $v' = \texttt{\{}^{X} dvs_{|x}\ x\texttt{\}}$, otherwise $\texttt{\{}^{X} dvs_{|x}\ x\texttt{\}} \cong x$ applying rule (block-elim), so $v' \cong \texttt{\{}^{X} dvs\ x\texttt{\}}$, which proves the result.
<u>Case: u is a block</u>. By induction hypothesis $u \cong \texttt{\{}^{Y} dvs'\ y\texttt{\}}$, and $dvs'_{|y} = dvs'$. Therefore, $v \cong \texttt{\{}^{X} dvs\ \texttt{\{}^{Y} dvs'\ y\texttt{\}}\texttt{\}}$. We first apply rule (alpha) to rename the variables declared in dvs' to be different from the bound and free variables of dvs, producing $\texttt{\{}^{Y'} dvs''\ y'\texttt{\}}$. Then rule (body) can be applied to obtain $v \cong \texttt{\{}^{X \cup Y'} dvs\ dvs''\ \texttt{\{}^{\emptyset}\ y'\texttt{\}}\texttt{\}}$ and with rule (block-elim) we obtain $v \cong \texttt{\{}^{X \cup Y'} dvs\ dvs''\ y'\texttt{\}}$. Finally applying rule (garbage) as done for the base case of the induction we have that: $v \cong \texttt{\{}^{X \cup Y'} dvs_1\ y'\texttt{\}}$, where $dvs_{1|y'} = dvs_1$. ◀

Values of the shape $\texttt{\{}^{X} dvs\ x\texttt{\}}$, such that $\mathsf{FV}(\texttt{\{}^{X} dvs_{|x}\ x\texttt{\}}) = \emptyset$ are capsules. Their inner objects can be reached only through a reference to the block.

We introduce now some notations which will be used in reduction rules. We write $\mathit{dvs}(x)$ for *the declaration of x in dvs*, if any (recall that in well-formed blocks there are no multiple declarations for the same variable).

We write $\mathsf{HB}(\mathcal{E})$ for the *hole binders* of $\mathcal{E}$, that is, the variables declared in blocks enclosing the context hole, defined by:

- if $\mathcal{E} = \texttt{\{}\mathit{dvs}\ T\,x\texttt{=}\mathcal{E}';\ \mathit{ds}\ e\texttt{\}}$, then $\mathsf{HB}(\mathcal{E}) = \mathsf{dom}(\mathit{dvs}) \cup \{x\} \cup \mathsf{HB}(\mathcal{E}')$
- if $\mathcal{E} = \texttt{\{}\mathit{dvs}\ \mathcal{E}'\texttt{\}}$, then $\mathsf{HB}(\mathcal{E}) = \mathsf{dom}(\mathit{dvs}) \cup \mathsf{HB}(\mathcal{E}')$

We write $\mathcal{E}_x$ and $\mathsf{dec}(\mathcal{E}, x)$ for the *sub-context including of the declaration of x* and the *declaration of x* extracted from $\mathcal{E}$, defined as follows:

- let $\mathcal{E} = \texttt{\{}\mathit{dvs}\ T\,y\texttt{=}\mathcal{E}';\ \mathit{ds}\ e\texttt{\}}$;
  - if $\mathit{dvs}(x) = \mathit{dv}$ and $x \notin \mathsf{HB}(\mathcal{E}')$, then $\mathcal{E}_x = \texttt{\{}\mathit{dvs}\ T\,y\texttt{=}[\,];\ \mathit{ds}\ e\texttt{\}}$ and $\mathsf{dec}(\mathcal{E}, x) = \mathit{dv}$
  - else $\mathcal{E}_x = \texttt{\{}\mathit{dvs}\ T\,y\texttt{=}\mathcal{E}'_x;\ \mathit{ds}\ e\texttt{\}}$ and $\mathsf{dec}(\mathcal{E}, x) = \mathsf{dec}(\mathcal{E}', x)$
- let $\mathcal{E} = \texttt{\{}\mathit{dvs}\ \mathcal{E}'\texttt{\}}$;
  - if $\mathit{dvs}(x) = \mathit{dv}$ and $x \notin \mathsf{HB}(\mathcal{E}')$, then $\mathcal{E}_x = \texttt{\{}\mathit{dvs}\ [\,]\texttt{\}}$ and $\mathsf{dec}(\mathcal{E}, x) = \mathit{dv}$
  - else $\mathcal{E}_x = \texttt{\{}\mathit{dvs}\ \mathcal{E}'_x\texttt{\}}$, and $\mathsf{dec}(\mathcal{E}, x) = \mathsf{dec}(\mathcal{E}', x)$

Note that $\mathcal{E}_x$ and $\mathsf{dec}(\mathcal{E}, x)$ are not defined if x is not declared in any block enclosing the context hole, that is, if $x \notin \mathsf{HB}(\mathcal{E})$.

Finally, we write $e[v/x]$ for the expression obtained by replacing all (free) occurrences of x in e by v, defined in the standard way.

Reduction rules are given in Fig.8.

$$\textsc{(congr)}\ \frac{e_1 \longrightarrow e_2}{e'_1 \longrightarrow e'_2} \quad \begin{array}{l} e_1 \cong e'_1 \\ e_2 \cong e'_2 \end{array}$$

$$\textsc{(field-access)}\ \mathcal{E}[x\,\texttt{.}f_i] \longrightarrow \mathcal{E}[x_i] \qquad \begin{array}{l} \mathsf{dec}(\mathcal{E}, x) = C\ x\texttt{=}\mathtt{new}\ C\texttt{(}x_1, \ldots, x_n\texttt{);} \\ \mathcal{E} = \mathcal{E}_x[\mathcal{E}'] \quad \wedge \quad x_i \notin \mathsf{HB}(\mathcal{E}') \\ \mathsf{fields}(C) = C_1\ f_1 \ldots C_n\ f_n \end{array}$$

$$\textsc{(met-call)}\ \begin{array}{l} \mathcal{E}[v\,\texttt{.}m\texttt{(}v_1, .., v_n\texttt{)}] \longrightarrow \\ \mathcal{E}[\texttt{\{}C^{\mu}\,\mathtt{this}\texttt{=}v;\ T_1\,x_1\texttt{=}v_1; .. T_n\,x_n\texttt{=}v_n;\ e\texttt{\}}] \end{array} \qquad \begin{array}{l} \mathsf{class}(\mathcal{E}, v) = C \\ \mathsf{meth}(C, m) = \langle\_, \mu, T_1\ x_1 .. T_n\ x_n, e\rangle \end{array}$$

$$\textsc{(field-assign)}\ \mathcal{E}[x\,\texttt{.}f_i\texttt{=}y] \longrightarrow \mathcal{E}^{x=v}[y] \qquad \begin{array}{l} \mathsf{dec}(\mathcal{E}, x) = C\ x\texttt{=}\mathtt{new}\ C\texttt{(}x_1, \ldots, x_i, \mathit{xs}\texttt{);} \\ \mathcal{E} = \mathcal{E}_x[\mathcal{E}'] \quad \wedge \quad y \notin \mathsf{HB}(\mathcal{E}') \\ \mathsf{fields}(C) = C_1\ f_1 \ldots C_n\ f_n \\ v = \mathtt{new}\ C\texttt{(}x_1, \ldots, x_{i-1}, y, \mathit{xs}\texttt{)} \end{array}$$

$$\textsc{(alias-elim)}\ \mathcal{E}[\texttt{\{}^X \mathit{dvs}\ C\,x\texttt{=}y;\ \mathit{ds}\ e\texttt{\}}] \longrightarrow \mathcal{E}[\texttt{\{}^X \mathit{dvs}\ \mathit{ds}\ e\texttt{\}}[y/x]]$$

$$\textsc{(affine-elim)}\ \mathcal{E}[\texttt{\{}\mathit{dvs}\ C^{\mathsf{a}}\,x\texttt{=}v;\ \mathit{ds}\ e\texttt{\}}] \longrightarrow \mathcal{E}[\texttt{\{}\mathit{dvs}\ \mathit{ds}\ e\texttt{\}}[v/x]]$$

Figure 8 Reduction rules

Rule (congr) states that congruence can be used to reduce a term which otherwise would be stuck, as it happens for α-rule in lambda calculus.

In rule (field-access), given a field access of shape $x\,\texttt{.}f$, the first enclosing declaration for x is found (through the auxiliary function dec). Since $\mathsf{dec}(\mathcal{E}, x)$ is defined, also $\mathcal{E}_x$ is defined, identifying the sub-context $\mathcal{E}'$, such that $\mathcal{E}'[x\,\texttt{.}f]$ is contained in the block of the declaration of x. The 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 field access is reduced to the reference x_i stored in this field. The side condition $x_i \notin \mathsf{HB}(\mathcal{E}')$ ensures that there are no inner declarations for x_i (otherwise x_i would be erroneously bound). This can be always obtained by rule (congr) using rule (alpha) of Fig.7.

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}
```

In rule (INVK), the class C of the receiver v is found (through the auxiliary function *class*, whose formal definition is omitted), and method m of C, if any, is retrieved from the class table. In this case, the call is reduced to a block where declarations of the appropriate type for `this` and the parameters are initialized with the receiver and the arguments, respectively.

In rule (FIELD-ASSIGN), given a field assignment of shape x.f=y, the first enclosing declaration for x is found (through the auxiliary function dec). 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 $\mathcal{E}^{u=\mathcal{E}}[x]$ for the evaluation context obtained from $\mathcal{E}$ by replacing the right-hand side of the declaration of x by u (the obvious formal definition is omitted). As for rule (FIELD-ACCESS) we have the side condition that $y \notin \mathsf{HB}(\mathcal{E}')$. This side condition, requiring that there are no inner declarations for the reference y, prevents scope extrusion, since if $y \in \mathsf{HB}(\mathcal{E}')$, $\mathcal{E}^{u=u}[x]$ would take y outside the scope of its definition. The congruence rule of Fig.7 can be used to *correctly* move the declaration of y outside its declaration block, as previously described. For example, without the side condition, the term (without annotations)

```
D x=new D(...); {C y=new C(); x.f=y}
```

would reduce to

```
D x=new D(y);  {C y=new C();  y}
```

The previous term is congruent to

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

by applying rule (BODY), and then (BLOCK-ELIM). This term reduces correctly to

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

The last two rules eliminate evaluated declarations from a block.

In rule (ALIAS-ELIM), a reference (non affine variable) x which is initialized as an alias of another reference y is eliminated by replacing all its occurrences. In rule (AFFINE-ELIM), an affine variable is eliminated by replacing its unique occurrence.

6 Results

In this section we present the main formal results of our calculus. In particular, we show the soundness of the type system for the operational semantics. We also give a refined version of subject reduction, Theorem 15, showing that reduction preserves type and sharing relation of subexpressions, so that capsule subexpressions reduce to closed values. This is significant since the congruence relation may change the block structure of terms.

The following definition introduces the type judgement for annotated expressions derived from the one of their underlined (user-level) expression, and extends the judgement to sequences of declarations.

▶ **Definition 7.** ▬ Let e be an annotated expression, the judgement $\Gamma \vdash e : C \mid X \mid \mathcal{S}$ stands for $\Gamma \vdash e^- : C \mid X \mid \mathcal{S} \rightsquigarrow e$ where e^- is obtained by erasing the annotations from e.

▬ Let ds be $C_1^{\mu_1}\ x_1$=e_1;... $C_n^{\mu_n}\ x_n$=e_n; where e_i are annotated expressions, $\Gamma \vdash d_1 \cdots d_n : \mathcal{S}$, stands for,

1. $\Gamma \vdash e_i : C_i \mid X_i \mid \mathcal{S}_i$, for some X_i, and $\mathcal{S}_i$ $(1 \leq i \leq n)$,
2. $\mathcal{S} = \sum_{i=1}^{n} \mathcal{S}_i + (\{x_i\} \cup X_i)$, and
3. if μ_i=a then X_i=$\emptyset$ $(1 \leq i \leq n)$.

Given Γ, let $\Gamma \setminus x$ be the type environment obtained by removing the type association for x from Γ, if there is one. Typing depends only on the free variables of the expression.

▶ **Lemma 8** (weakening). *Let $\Gamma \vdash e : C \mid X \mid \mathcal{S} \rightsquigarrow e'$. If $x \notin \mathsf{FV}(e)$, then*
1. $\Gamma[x{:}T] \vdash e : C \mid X \mid \mathcal{S} \rightsquigarrow e'$ *for all T, and*
2. $\Gamma \setminus x \vdash e : C \mid X \mid \mathcal{S} \rightsquigarrow e'$.

Proof. By induction on derivations. ◀

Let ds be $T_1\ x_1$=e_1;... $T_n\ x_n$=e_n; where the e_i are annotated expressions, *the type environment associated to ds*, Γ_{ds} is $x_1{:}T_1, \ldots, x_n{:}T_n$.

The following lemma asserts that congruent expressions have the same type, produce results which are connected to the same variables, and induce sharing relations that coincide on the free variables of both expressions.

▶ **Lemma 9.** *Let e_1 and e_2 be annotated terms such that $\Gamma \vdash e_1 : C_1 \mid X_1 \mid \mathcal{S}_1$ and $\Gamma \vdash e_2 : C_2 \mid X_2 \mid \mathcal{S}_2$. If $e_1 \cong e_2$, then $C_1 = C_2$, $X_1 = X_2$, and $\mathcal{S}_1 = \mathcal{S}_2$.*

Proof. By cases on the congruence rule used. We do the most interesting cases, which are rule (DEC), and (GARBAGE) of Fig.7.

Rule (DEC). In this case

1. $e_1 = \{^{Y} ds\ C^{\mu}\ x$=$\{^{X} ds_1\ ds_2\ e\};\ ds'\ e'\}$,
2. $e_2 = \{^{Y} ds\ ds_1\ C^{\mu}\ x$=$\{^{X \setminus \mathsf{dom}(ds_1)} ds_2\ e\};\ ds'\ e'\}$,
3. $\mathsf{FV}(ds_1) \cap \mathsf{dom}(ds_2) = \emptyset$,
4. $\mathsf{FV}(ds\, ds'\, e') \cap \mathsf{dom}(ds_1) = \emptyset$, and
5. if $\mu = \mathsf{a}$ then $\mathsf{dom}(ds_1) \cap X = \emptyset$.

Let $\Gamma \vdash e_1 : C_1 \mid X_1 \mid \mathcal{S}_1$, define $\Gamma_1 = \Gamma[\Gamma_{ds}, x{:}C^{\mu}, \Gamma_{ds'}]$, and $Z_d = \mathsf{dom}(ds) \cup \mathsf{dom}(ds') \cup \{x\}$. From 1. and rule (T-BLOCK) we have that

(a) $\Gamma_1 \vdash e' : C_1 \mid X' \mid \mathcal{S}'$ for some X' and $\mathcal{S}'$, and
(b) $\Gamma_1 \vdash ds\ C^{\mu}\ x$=$\{^{X} ds_1\ ds_2\ e\};\ ds' : \mathcal{S}_d$, where $\mathcal{S}_d = \mathcal{S}_{ds} + \mathcal{S}_x + \mathcal{S}_{ds'}$ for some $\mathcal{S}_{ds}$, $\mathcal{S}_x$, and $\mathcal{S}_{ds'}$.

Therefore, $Y = [X']_{(\mathcal{S}_d + \mathcal{S}')} \cap Z_d$, $X_1 = [X']_{(\mathcal{S}_d + \mathcal{S}')} \setminus Z_d$, and $\mathcal{S}_1 = (\mathcal{S}_d + \mathcal{S}') \setminus Z_d$. From (b) we have that $\Gamma_1 \vdash \{^{X} ds_1\ ds_2\ e\} : C \mid Y_x \mid \mathcal{S}_x$, for some Y_x, and $\mathcal{S}_x$, and if $\mu = \mathsf{a}$, then $Y_x = \emptyset$. Let $\Gamma_1' = \Gamma_1'[\Gamma_{ds_1}, \Gamma_{ds_2}]$, $\mathcal{S}_x' = \mathcal{S}_{ds_1} + \mathcal{S}_{ds_2} + \mathcal{S}$, and $Z = \mathsf{dom}(ds_1) \cup \mathsf{dom}(ds_2)$ from rule (T-BLOCK)

(c) $\Gamma_1' \vdash e : C \mid X'' \mid \mathcal{S}$ for some X'' and $\mathcal{S}$,
(d) $\Gamma_1' \vdash ds_1 : \mathcal{S}_{ds_1}$ for some $\mathcal{S}_{ds_1}$,
(e) $\Gamma_1' \vdash ds_2 : \mathcal{S}_{ds_2}$ for some $\mathcal{S}_{ds_2}$, and
(f) $\mathcal{S}_x = \mathcal{S}_x' \setminus Z$, $Y_x = [X'']_{\mathcal{S}_x'} \setminus Z$, and $X = [X'']_{\mathcal{S}_x'} \cap Z$.

From 4., we may assume that $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(ds_1) = \emptyset$. So $\Gamma_1' = \Gamma_1, \Gamma_{ds_1}[\Gamma_{ds_2}]$. Let $\mathcal{S}_x'' = \mathcal{S}_{ds_2} + \mathcal{S}$. From (c) and (e) applying rule (T-BLOCK) we have that

(∗) $\Gamma_1, \Gamma_{ds_1} \vdash \{^{X\backslash \mathsf{dom}(ds_1)} ds_2\ e\} : C \mid [X'']_{\mathcal{S}''_x} \backslash \mathsf{dom}(ds_2) \mid \mathcal{S}''_x \backslash \mathsf{dom}(ds_2)$.
Assume that $\mu = \mathsf{a}$. Since $\mathcal{S}'_x = \mathcal{S}''_x + \mathcal{S}_{ds_1}$, we have that $[X'']_{\mathcal{S}''_x} \subseteq [X'']_{\mathcal{S}'_x}$. Moreover, from $Y_x = [X'']_{\mathcal{S}'_x} \backslash Z = \emptyset$, and 5. we get that $[X'']_{\mathcal{S}'_x} \backslash \mathsf{dom}(ds_2) = \emptyset$, and therefore also $[X'']_{\mathcal{S}''_x} \backslash \mathsf{dom}(ds_2) = \emptyset$.
Define $\Gamma''_1 = \Gamma[\Gamma_{ds}, \Gamma_{ds_1}, x{:}C^\mu, \Gamma_{ds'}]$. From 4. we have that $\Gamma''_1 = \Gamma_1, \Gamma_{ds_1}$. From (a) and Lemma 8.1, we derive
(a1) $\Gamma''_1 \vdash e' : C_1 \mid X' \mid \mathcal{S}'$.
From (d), 3., 4., and Lemma 8.2, we have that $\Gamma''_1 \vdash ds_1 : \mathcal{S}_{ds_1}$. Therefore, from (b), (∗), and Lemma 8.1, we derive
(b1) $\Gamma''_1 \vdash ds\ ds_1\ C^\mu\ x\texttt{=}\{^{X\backslash \mathsf{dom}(ds_1)} ds_2\ e\};\ ds' : \mathcal{S}'_d$, where $\mathcal{S}'_d = \mathcal{S}_{ds} + \mathcal{S}_{ds_1} + (\mathcal{S}''_x \backslash \mathsf{dom}(ds_2)) + \mathcal{S}_{ds'}$.
Let $Z' = \mathsf{dom}(ds) \cup \mathsf{dom}(ds') \cup \mathsf{dom}(ds_1) \cup \{x\}$. From (a1), (b1) and rule (T-BLOCK) we get

$$\Gamma \vdash \{^{Y'} ds\ ds_1\ C^\mu\ x\texttt{=}\{^{X\backslash \mathsf{dom}(ds_1)} ds_2\ e\};\ ds'\ e'\} : C_1 \mid X'_1 \mid \mathcal{S}'_1$$

where, $Y' = [X']_{(\mathcal{S}'_d+\mathcal{S}')} \cap Z'$, $X'_1 = [X']_{(\mathcal{S}'_d+\mathcal{S}')} \backslash Z'$ and $\mathcal{S}'_1 = (\mathcal{S}'_d + \mathcal{S}') \backslash Z'$. Let $Z' = \mathsf{dom}(ds) \cup \mathsf{dom}(ds') \mathsf{dom}(ds_1) \cup \{x\}$. For 4., and Proposition 1.1 and 3, we have that the domain of $\mathcal{S}_{ds} + (\mathcal{S}''_x \backslash \mathsf{dom}(ds_2)) + \mathcal{S}_{ds'}$ and X'' do not contain variables in $\mathsf{dom}(ds_1)$. Therefore, $Y' = Y$, $X'_1 = X_1$, and $\mathcal{S}'_1 + \mathcal{S}_1$. Therefore, $\Gamma \vdash e_2 : C_1 \mid X_1 \mid \mathcal{S}_1$.
In a similar way we can prove that $\Gamma \vdash e_2 : C_2 \mid X_2 \mid \mathcal{S}_2$ implies $\Gamma \vdash e_1 : C_2 \mid X_2 \mid \mathcal{S}_2$, which proves the result.

Rule (GARBAGE). In this case
1. $e_1 = \{^X dvs\ ds\ e\}$,
2. $e_2 = \{^{X\backslash \mathsf{dom}(dvs)} ds\ e\}$,
3. $(\mathsf{FV}(ds) \cup \mathsf{FV}(e)) \cap \mathsf{dom}(dvs) = \emptyset$.

From 3., and Definition 1 we have that, $dvs_{|(\mathsf{FV}(e) \cup \mathsf{FV}(ds))} = \epsilon$. Therefore, from Proposition 1.4, we have that $X = X \setminus \mathsf{dom}(dvs)$.
Let $\Gamma \vdash e_1 : C_1 \mid X_1 \mid \mathcal{S}_1$, from 1. and rule (T-BLOCK) we have that
(a) $\Gamma[\Gamma_{dvs}, \Gamma_{ds}] \vdash e : C_1 \mid Y \mid \mathcal{S}$ for some Y and $\mathcal{S}$,
(b) $\Gamma[\Gamma_{dvs}, \Gamma_{ds}] \vdash ds : \mathcal{S}'$ for some $\mathcal{S}'$, and
(c) $\Gamma[\Gamma_{dvs}, \Gamma_{ds}] \vdash dvs : \mathcal{S}''$ for some $\mathcal{S}''$.
Let $Z = \mathsf{dom}(dvs) \cup \mathsf{dom}(ds)$, we have that $X = [Y]_{(\mathcal{S}+\mathcal{S}'+\mathcal{S}'')} \cap Z$, $X_1 = [Y]_{(\mathcal{S}+\mathcal{S}'+\mathcal{S}'')} \backslash Z$, and $\mathcal{S}_1 = (\mathcal{S} + \mathcal{S}' + \mathcal{S}'') \backslash Z$. From 3., (a), (b), and Lemma 8.2, we have that
(a1) $\Gamma[\Gamma_{ds}] \vdash e : C_1 \mid Y \mid \mathcal{S}$, and
(b1) $\Gamma[\Gamma_{ds}] \vdash ds : \mathcal{S}'$.
From (a1), (b1), and rule (T-BLOCK) we get $\Gamma \vdash \{^X ds\ e\} : C_1 \mid X'_1 \mid \mathcal{S}'_1$ where $X'_1 = [Y]_{(\mathcal{S}'+\mathcal{S})} \backslash \mathsf{dom}(ds)$ and $\mathcal{S}'_1 = (\mathcal{S}' + \mathcal{S}) \backslash \mathsf{dom}(ds)$. From 3., and Proposition 1.1 and 3, we have that the domain of $\mathcal{S}' + \mathcal{S}$ and Y do not contain variables in $\mathsf{dom}(dvs)$. Therefore, $X'_1 = X_1$, and $\mathcal{S}'_1 = \mathcal{S}_1$. Therefore, $\Gamma \vdash e_2 : C_1 \mid X_1 \mid \mathcal{S}_1$.
On the other side, assume that $\Gamma \vdash e_2 : C_2 \mid X_2 \mid \mathcal{S}_2$, from 2., and rule (T-BLOCK) we have that
(a2) $\Gamma[\Gamma_{ds}] \vdash e : C_2 \mid Y \mid \mathcal{S}$ for some Y and $\mathcal{S}$,
(b2) $\Gamma[\Gamma_{ds}] \vdash ds : \mathcal{S}'$ for some $\mathcal{S}'$.
where $X_2 = [Y]_{(\mathcal{S}+\mathcal{S}')} \backslash \mathsf{dom}(ds)$, and $\mathcal{S}_2 = (\mathcal{S} + \mathcal{S}') \backslash \mathsf{dom}(ds)$. Since we have that $\Gamma \vdash e_1 : C_1 \mid X_1 \mid \mathcal{S}_1$, from rule (T-BLOCK),
(c2) $\Gamma[\Gamma_{dvs}, \Gamma_{ds}] \vdash dvs : \mathcal{S}''$ for some $\mathcal{S}''$.
From 3., (a2), (b2), and Lemma 8.1, we have that
(a3) $\Gamma[\Gamma_{dvs}, \Gamma_{ds}] \vdash e : C_2 \mid Y \mid \mathcal{S}$, and
(b3) $\Gamma[\Gamma_{dvs}, \Gamma_{ds}] \vdash ds : \mathcal{S}'$.

Therefore, from (a3), (b3), (c2), and rule (T-BLOCK) we get $\Gamma \vdash \{^{X} dvs\ ds\ e\} : C_2 \mid X'_2 \mid \mathcal{S}'_2$, where $X'_2 = [Y]_{(\mathcal{S}+\mathcal{S}'+\mathcal{S}'')}\backslash(\mathsf{dom}(ds) \cup \mathsf{dom}(dvs))$, and $\mathcal{S}'_2 = (\mathcal{S} + \mathcal{S}' + \mathcal{S}'')\backslash(\mathsf{dom}(ds) \cup \mathsf{dom}(dvs))$. From 3., and Proposition 1.1 and 3, we have that the domain of $\mathcal{S}' + \mathcal{S}$ and Y do not contain variables in $\mathsf{dom}(dvs)$. Therefore, $X'_2 = X_2$ and $\mathcal{S}'_2 = \mathcal{S}_2$, which proves the result. ◀

To prove subject reduction and progress, we introduce the set of redexes, and show that, expressions can be decomposed in a unique way in an evaluation context filled with a redex. So that, an expression is either a value or it matches the left-hand-side of exactly one reduction rule.

▶ **Definition 10.** Redexes, ρ, are defined by:

$$\rho ::= x.f_i \mid v.m(v_1,\dots,v_n) \mid x.f{=}y \mid \{^{X} dvs\ C\, x{=}y;\ ds\ e\} \mid \{dvs\ C^{\mathsf{a}}\, x{=}v;\ ds\ e\}$$

▶ **Lemma 11** (Unique Decomposition). *If e is not a value, then there are $\mathcal{E}$, and ρ such that $e \cong \mathcal{E}[\rho]$. Moreover, ρ is unique and $\mathcal{E}$ is unique up to congruence.*

Proof. By structural induction on expressions. We show some cases.

Case $\underline{e.f}$. If e is not a value, then by induction hypothesis, there are $\mathcal{E}$, and ρ such that $e \cong \mathcal{E}[\rho]$. Let $\mathcal{E}' = \mathcal{E}.f$, we have that $e.f \cong \mathcal{E}'[\rho]$. Moreover, $\mathcal{E}'$, is unique up to congruence. If e is a value, and $e = x$, then the result holds, since $x.f$ is a redex, and with $\mathcal{E} = [\ ]$ we have that $e.f = \mathcal{E}[x.f]$.
If $e \cong \{^{X} dvs'\ u\}$, by Lemma 2, $e \cong \{^{X} dvs\ x\}$ then $e.f = \{^{X} dvs\ x\}.f$, and $\{^{X} dvs\ x\}.f \cong \{^{X} dvs\ x.f\}$. Let $\mathcal{E} = \{^{X} dvs\ [\ ]\}$, and $\rho = x.f$, $e.f \cong \mathcal{E}[\rho]$.

Case $\underline{\texttt{new}\ C(es)}$. If for some i, $1 \leq i \leq n$, e_i is not a value, by induction hypothesis, there are $\mathcal{E}$, and ρ such that $e_i \cong \mathcal{E}[\rho]$ using $\mathcal{E}' = \texttt{new}\ C(vs, \mathcal{E}, es)$ we have that $\texttt{new}\ C(es) \cong \mathcal{E}'[\rho]$. If we have $\texttt{new}\ C(xs, v_1, \dots, v_n)$, and v_1 is not a variable, then since by Lemma 2 $v_1 \cong \{^{Y_1} dvs_1\ y_1\}$, applying congruence rule (NEW-ARG) we have
$\texttt{new}\ C(xs, vs) \cong \{^{Y_1\backslash\mathsf{dom}(dvs_1)} dvs_1\ \texttt{new}\ C(xs, y_1, v_2 \dots, v_n)\}$. Iterating the application of rule (NEW-ARG), followed by rule (BODY), we obtain that $\texttt{new}\ C(es) \cong \{^{X} dvs\ \texttt{new}\ C(xs, ys)\}$ for some dvs, ys, and X. Finally, applying rule (NEW), $\texttt{new}\ C(es) \cong e'$, where
$e' = \{^{X\cup\{x\}} dvs\ C\, x{=}\texttt{new}\ C(xs, ys);\ x\}$. Since e' is a value the result is proved.

Case $\underline{\{^{X} ds\ e\}}$. If the block is not a value then either $ds = dvs\ T\, x{=}e_1;\ ds_1$ where e_1 is not a value or $ds = dvs$ and e is not a value. In the first case, by induction hypothesis, there are $\mathcal{E}$, and ρ such that $e' \cong \mathcal{E}[\rho]$. Consider the block $bl = \{^{X} dvs\ dvs_1\, T\, x{=}e';\ ds_2\ e\}$ where dvs_1 are all the evaluated declarations of ds_1. Applying rule (REORDER) of Fig.7, $bl \cong \{^{X} dvs\ T\, x{=}e_1;\ ds_1\ e\}$. Let $\mathcal{E}' = \{^{X} dvs\ dvs_1\, T\, x{=}\mathcal{E};\ ds_2\ e\}$, we have that $\{^{X} ds\ e\} \cong \mathcal{E}'[\rho]$.
If the expression is $\{^{X} dvs\ e\}$, and e is not a value, by induction hypothesis, there are $\mathcal{E}$, and ρ such that $e \cong \mathcal{E}[\rho]$. The context $\mathcal{E}' = \{^{X} dvs\ \mathcal{E}\}$ proves the result.
Otherwise the expression is $\{^{X} dvs\ v\}$, which is a value. ◀

Given an evaluation context $\mathcal{E}$, let $\Gamma_{\mathcal{E}}$ be the *type environment extracted from* $\mathcal{E}$, defined by:

- if $\mathcal{E} = \{dvs\ T\, x{=}\mathcal{E}';\ ds\ e\}$, then $\Gamma_{\mathcal{E}} = \Gamma_{dvs}, x{:}T, \Gamma_{ds}[\Gamma_{\mathcal{E}'}]$,
- if $\mathcal{E} = \{dvs\ \mathcal{E}'\}$, then $\Gamma_{\mathcal{E}} = \Gamma_{dvs}[\Gamma_{\mathcal{E}'}]$, and
- for all other contexts $\Gamma_{\mathcal{E}} = \Gamma_{\mathcal{E}'}$, where $\mathcal{E}'$ is the (unique) sub-context of $\mathcal{E}$.

The following lemma asserts that, subexpressions of typable expressions are themselves typable, and may be replaced with expressions that have the same type, but could have a less sharing and whose result may be connected to less variables.

▶ **Lemma 12** (Context). *Let $\Gamma \vdash \mathcal{E}[e] : C \mid X \mid \mathcal{S}$, then*

1. $\Gamma[\Gamma_{\mathcal{E}}] \vdash e : D \mid Y \mid \mathcal{S}_1$ *for some* D*,* Y *and* $\mathcal{S}_1$*,*
2. *if* $\Gamma[\Gamma_{\mathcal{E}}] \vdash e' : D \mid Y' \mid \mathcal{S}'$*, where* $Y' \subseteq Y$ *and* $\mathcal{S}' \sqsubseteq \mathcal{S}_1$*, then* $\Gamma \vdash \mathcal{E}[e'] : C \mid X' \mid \mathcal{S}''$ *such that* $X' \subseteq X$ *and* $\mathcal{S}'' \sqsubseteq \mathcal{S}$*.*

Proof. By induction on evaluation contexts. ◀

▶ **Lemma 13** (Substitution). *1. Let* $\Gamma, x{:}D^{a} \vdash e : C \mid X \mid \mathcal{S}$*, and* $\Gamma \vdash v : D \mid \emptyset \mid \epsilon$*. Then* $\Gamma \vdash e[v/x] : C \mid X \mid \mathcal{S}$*.*
2. Let $\Gamma, x{:}D \vdash e : C \mid X \mid \mathcal{S}$*, and* $\Gamma \vdash v : D \mid Y - \{x\} \mid \mathcal{S}$*. Then* $\Gamma \vdash e[v/x] : C \mid X \mid \mathcal{S}$*.*

Reduction preserves types but may produce expressions whose results have less connections, and yield less sharing between variables.

▶ **Theorem 14** (Subject Reduction). *If* $\Gamma \vdash e_1 : C \mid X \mid \mathcal{S}$*, and* $e_1 \longrightarrow e_2$*, then* $\Gamma \vdash e_2 : C \mid X' \mid \mathcal{S}'$ *where* $X' \subseteq X$*, and* $\mathcal{S}' \sqsubseteq \mathcal{S}$*.*

Proof. By Lemma 11, $e_1 \cong \mathcal{E}[\rho]$. Since $e_1 \longrightarrow e_2$, then $\mathcal{E}[\rho] \longrightarrow \mathcal{E}'[e']$, where $\mathcal{E}' = \mathcal{E}$ for all the rules applied except for (FIELD-ASSIGN). From Lemma 12.1, and $\Gamma \vdash e_1 : C \mid X \mid \mathcal{S}$, we have that $\Gamma[\Gamma_{\mathcal{E}}] \vdash \rho : D \mid Y \mid \mathcal{S}_\rho$ for some D, Y and $\mathcal{S}_\rho$. By case analysis on redexes, we can prove that the expression, e, such that $\mathcal{E}[e] = e_2$ is such that $\Gamma[\Gamma_{\mathcal{E}}] \vdash e : D \mid Y_1 \mid \mathcal{S}_1$, where $Y_1 \subseteq Y$ and $\mathcal{S}_1 \sqsubseteq \mathcal{S}_\rho$. For alias and affine elimination, the result follows by Lemma 13. Therefore, for all the rules except for (FIELD-ASSIGN), by Lemma 12.2 we derive that $\Gamma \vdash e_2 : C \mid X' \mid \mathcal{S}'$, where $X' \subseteq X$, and $\mathcal{S}' \sqsubseteq \mathcal{S}$.
Consider rule (FIELD-ASSIGN). It is easy to see that $\Gamma_{\mathcal{E}} = \Gamma_{\mathcal{E}'}$ since the only change in the evaluation contexts is the update of the field of x with a reference with the correct type for the field. So, from Lemma 12.2, we derive the result. ◀

In addition to soundness, we state that expressions whose type has a capsule modifier actually ensures the expected behaviour. A nice consequence of our non standard operational model is that this can be easily expressed and proved, as shown below, since a capsule is simply a closed value.

▶ **Theorem 15** (Capsule). *Let* $\mathcal{E}$ *and* e *be such that* $\Gamma_{\mathcal{E}} \vdash e : C \mid X \mid \mathcal{S}$*. If* $\mathcal{E}[e] \longrightarrow^{\star} \mathcal{E}'[v]$*, where* $\Gamma_{\mathcal{E}} = \Gamma_{\mathcal{E}'}$*, then* $\Gamma_{\mathcal{E}} \vdash e' : C \mid X' \mid \mathcal{S}'$ *where* $X' \subseteq X$*, and* $\mathcal{S}' \sqsubseteq \mathcal{S}$*. Therefore, if* $X = \emptyset$*, then* $v \cong$ {*dvs x*}*, such that* $\mathsf{FV}(${*dvs x*}$) = \emptyset$*.*

Proof. The proof is similar to the one of Subject Reduction. ◀

Define $\mathcal{H}_{\mathcal{E}}$ for $\mathcal{E} \neq [\,]$ to the *block containing the hole of* $\mathcal{E}$. That is,

- Let $\mathcal{E} = ${*dvs T y*=$\mathcal{E}'$; *ds e*}; if $\mathcal{E}' = [\,]$ then $\mathcal{H}_{\mathcal{E}} = \mathcal{E}$, else $\mathcal{H}_{\mathcal{E}} = \mathcal{H}_{\mathcal{E}'}$.
- Let $\mathcal{E} = ${*dvs* $\mathcal{E}'$}; if $\mathcal{E}' = [\,]$ then $\mathcal{H}_{\mathcal{E}} = \mathcal{E}$, else $\mathcal{H}_{\mathcal{E}} = \mathcal{H}_{\mathcal{E}'}$.

Closed expressions are not “stuck”, so they are either values or reduce to some term.

▶ **Theorem 16** (Progress). *If* $\vdash e_1 : C \mid X \mid \mathcal{S}$*, and* e_1 *is not a value, then* $e_1 \longrightarrow e_2$ *for some* e_2*.*

Proof. By Lemma 11, if e_1 is not a value, then $e_1 \cong \mathcal{E}[\rho]$. For all ρ, except field access and field update, we have that the corresponding reduction rule is applicable, so $e_1 \longrightarrow e_2$ for some e_2.
If $\underline{\rho \text{ is } x.f_i}$, from Lemma 12.1, rule (T-FIELD-ACCESS), and rule (T-VAR) of Fig.3, we have that $\Gamma_{\mathcal{E}} \vdash x.f_i : C_i \mid \{x\} \mid \epsilon$ where $\mathsf{fields}(C) = C_1\ f_1 \ldots C_n\ f_n$. So, we have that $\mathcal{E}_x$ is defined, and, for some $\mathcal{E}'$, $\mathcal{E} = \mathcal{E}_x[\mathcal{E}']$. If $x_i \notin \mathsf{HB}(\mathcal{E}')$, then rule (FIELD-ACCESS) is applicable.

Otherwise, since $x_i \in \mathsf{HB}(\mathcal{E}')$, we have that $\mathcal{E}'_{x_i}$ is defined, and $\mathcal{E}'[x.f_i] = \mathcal{E}_1[\mathcal{H}_{\mathcal{E}'_{x_i}}[\mathcal{E}_2[x.f_i]]]$ for some $\mathcal{E}_1$ and $\mathcal{E}_2$ such that $\mathcal{H}_{\mathcal{E}'_{x_i}}[\mathcal{E}_2[x.f_i]] = \{^X dvs\ T\ x_i{=}v;\ ds\ e\}$. Using rule (alpha) of Fig.7 we have that $\{^X dvs\ T\ x_i{=}v;\ ds\ e\} \cong \{^{X[y/x_i]} dvs[y/x_i]\ \ T\ y{=}v[y/x_i];\ \ ds'[y/x_i]\ e[y/x_i]\}$ where y can be chosen such that $y \notin \mathsf{HB}(\mathcal{E}'')$. Therefore $\mathcal{E}_x[\mathcal{E}'[x.f_i]] \cong \mathcal{E}_x[\mathcal{E}_3[x.f_i]]$ where $x_i \notin \mathsf{HB}(\mathcal{E}_3)$. So $\mathcal{E}_x[\mathcal{E}_3[x.f_i]] \longrightarrow e_2$ by applying rule (field-access), and $\mathcal{E}_x[\mathcal{E}'[x.f_i]] \longrightarrow e_2$ by applying rule (congruence).

If ρ is $x.f_i{=}y$, from Lemma 12.1, rule (t-field-assign), and rule (t-var) of Fig.3, we have that $\Gamma_{\mathcal{E}} \vdash x.f_i{=}y : C_i \mid \{x,y\} \mid \{x,y\}$ where $\mathsf{fields}(C) = C_1\ f_1 \ldots C_n\ f_n$. So, we have that $\mathcal{E}_x$ is defined, and $\mathcal{E} = \mathcal{E}_x[\mathcal{E}']$ for some $\mathcal{E}'$. Therefore, for some $\mathcal{E}'_1$, $\mathcal{E} = \mathcal{E}'_1[\mathcal{H}_{\mathcal{E}_x}[\mathcal{E}']]$. If $y \notin \mathsf{HB}(\mathcal{E}')$, then rule (Field-Assign) is applicable.
Otherwise, since $y \in \mathsf{HB}(\mathcal{E}')$, we have that $\mathcal{E}'_y$ is defined, and $\mathcal{E}'[x.f_i{=}y] = \mathcal{E}_1[\mathcal{H}_{\mathcal{E}'_y}[\mathcal{E}_2[x.f_i{=}y]]]$ for some $\mathcal{E}_1$ and $\mathcal{E}_2$ such that $\mathcal{H}_{\mathcal{E}'_y}[\mathcal{E}_2[x.f_i{=}y]] = \{^Y dvs_{|\mathsf{FV}(v)}\ T\ y{=}v;\ ds\ e\}$.
By induction on the number $n > 0$ of blocks from which we have to extrude the declaration of y. Let $dvs_1 = T\ y{=}v;\ dvs_{|\mathsf{FV}(v)}$. If $n > 1$, then for some $\mathcal{E}'' \neq [\,]$,
(a) either $\{^X dvs'\ \ T\ x{=}v';\ \mathcal{E}''[\{^Y dvs_1\ ds\ e\}]\}$
(b) or $\{^X dvs'\ \ T\ x{=}v';\ \ T\ z{=}\mathcal{E}''[\{^Y dvs_1\ ds\ e\}];\ \ ds'\ e'\}$,
For (a), by induction hypothesis we have that,
- $\{^X dvs'\ \ T\ x{=}v';\ \mathcal{E}''[\{^Y dvs_1\ ds\ e\}]\} \cong \{^X dvs'\ \ T\ x{=}v';\ \{^{Y'} dvs_1\ ds'\ e'\}\}$ for some ds', e', and Y'.

Applying rule (Body) of Fig.7 we have that
- $\{^X dvs'\ \ T\ x{=}v';\ \{^{Y'} dvs_1\ ds'\ e'\}\} \cong \{^{X'} dvs'\ \ T\ x{=}v';\ \ dvs_1\ \{^{Y'\setminus\mathsf{dom}(dvs_1)} ds'\ e'\}\}$ where $X' = X \cup (Y' \cap \mathsf{dom}(dvs_1))$.

For (b), by induction hypothesis we have that,
- $\{^X dvs'\ \ T\ x{=}v';\ \ T\ z{=}\mathcal{E}''[\{^Y dvs_1\ ds\ e\}];\ \ ds'\ e'\} \cong \{^X dvs'\ \ T\ x{=}v';\ \ T\ z{=}\{^{Y'} dvs_1\ ds'\ e'\};\ \ ds''\ e''\}$ for some ds', ds'', e', e'', and Y'.

If $y \notin Y'$, from Proposition 1 we have that $\mathsf{dom}(dvs_{|\mathsf{FV}(val)}) \cap Y' = \emptyset$. Applying rule (Dec) of Fig.7 we get
- $\{^X dvs'\ \ T\ x{=}v';\ \ T\ z{=}\{^{Y'} dvs_1\ ds'\ e'\};\ \ ds''\ e''\} \cong \{^X dvs'\ \ T\ x{=}v';\ \ dvs_1\ \ T\ z{=}\{^{Y'} ds'\ e'\};\ \ ds''\ e''\}$.

If $y \in Y'$, from $\vdash e_1 : C \mid X \mid \mathcal{S}$, Lemma 12, $\Gamma_{\mathcal{E}_1} \vdash \{^Y dvs_1\ ds\ e\} : D \mid \{Z\} \mid \mathcal{S}$ for some D, Z, and $\mathcal{S}$ such that $T = D^\mu$. From $\Gamma_{\mathcal{E}} \vdash x.f_i{=}y : C_i \mid \{x,y\} \mid \{x,y\}$ and $y \in Y$ we also have that $x \in Z$. Therefore $\mu \neq \mathsf{a}$. Applying rule (Dec) we get
- $\{^X dvs'\ \ T\ x{=}v';\ \ T\ z{=}\{^{Y'} dvs_1\ ds'\ e'\};\ \ ds''\ e''\} \cong \{^X dvs'\ \ T\ x{=}v';\ \ dvs_1\ \ T\ z{=}\{^{Y'\setminus\mathsf{dom}(dvs_1)} ds'\ e'\};\ \ ds''\ e''\}$.

Therefore $\mathcal{E}_x[\mathcal{E}'[x.f_i{=}y]] \cong \mathcal{E}_x[\mathcal{E}_3[x.f_i{=}y]]$ for some $\mathcal{E}_3$ such that $y \notin \mathsf{HB}(\mathcal{E}_3)$. So $\mathcal{E}_x[\mathcal{E}_3[x.f_i{=}y]] \longrightarrow e_2$ by applying rule (field-assign), and $\mathcal{E}_x[\mathcal{E}'[x.f_i{=}y]] \longrightarrow e_2$ by applying rule (congruence). ◀

7 Related work

As mentioned in the Introduction, by the type and effect system in this paper we can express in a simple way and generalize two key notions which have been introduced in the literature on sharing and interference control.
- An expression has the *capsule* property if its evaluation returns an object graph that has no sharing with previously existing objects. We generalize this by always computing the set of references connected with the result, rather than just checking its emptiness.
- A reference is used as *lent* if the object graph it denotes can be updated, but not connected with objects which were previously disjoint. We generalize this to sets of references, rather than just singletons.

The capsule property has many variants in the literature, such as *isolated* [13], *external uniqueness* [8], *balloon* [1, 23], *island* [11], and the fact that aliasing can be controlled by using *lent* (*borrowed*) references is well-known [19]. However, before the work in [13], the capsule property was only detected in simple situations, such as using a primitive deep clone operator, or composing subexpressions with the same property.

The important novelty of the type system in [13] has been *recovery*, that is, the ability to detect properties (e.g., capsule or immutability) by keeping into account not only the expression itself but the way the surrounding context is used. Additional typing rules recognize special conditions on the context where the property can emerge. For instance, an expression which does not use external references at all is clearly a capsule, but this is the case also when used external references are all immutable.

In [24, 12] recovery has been improved, under the name of *promotion*, by including in the type system the *lent* notion as well: a capsule expression can use external mutable references as lent, and, moreover, lent references can be temporarily aliased, if all the other references are regarded as lent (*swapping*). Also the Pony language [9] builds on the recovery mechanisms of [13], but goes in a different direction, by distinguishing many different reference capabilities.

With respect to all these proposals, the type and effect system in this paper takes a drastically different approach: the (generalization of the) capsule and lent properties are *inferred* rather than imposed by declaring type modifiers. Whereas type inference has been sometimes included in works before [13], see, e.g., [3], this is at our knowledge the first time it is employed to express recovery/promotion. In this way, the desired properties emerge smoothly by normal bottom-up typechecking, and we obtain, roughly, the same expressive power of [24, 12], but in a much simpler and natural way. This can be seen as a radically simplified version of a region type system, as in [16].

A closed stream or research is that on *ownership* (see an overview in [7]) which, however, offers in a sense a specular approach. That is, a formal way is provided to express and prove the ownership invariant, which, however, is expected to be guaranteed by defensive cloning. In our approach, instead, the capsule concept models an efficient *ownership transfer*. In other words, when an object x is "owned" by another object y, it remains always true that y can be only accessed only through x, 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, as shown in the example at page 10.

Among the many works in this stream, we mention Rust [26], which uses ownership and type modifiers for memory management, and Kappa [6]. In the latter, types are compositions of one or more *capabilities*, and expose the union of their operations. The modes of the capabilities in a type control how resources of that type can be aliased. The compositional aspect of capabilities is an important difference from type modifiers, as accessing different parts of an object through different capabilities in the same type gives different properties.

Two other dimensions of comparison of our work with others are deep versus shallow approach and linearity versus destructive reads.

In our approach, as in [1, 12, 13, 23, 24], properties have a *deep* interpretation, in the sense that they are propagated to the whole reachable object graph. In a shallow interpretation, instead, as in [3, 4, 19, 20], it is possible, for instance, to reach a mutable object from an immutable object. In this sense, approaches based on ownership, or where it is somehow possible to have any form of "internal mutation" are shallow, as in [6, 14, 16, 26]. This also includes [8], where an unique object can point to arbitrarily shared objects, if they do not, in turn, point back to the unique object itself.

The advantage of the deep interpretation is that libraries can declare strong intentions in a coherent and uniform way, independently of the concrete representation of the user input (that, with the use of interfaces, could be unknown to the library). On the other side, providing (only) deep modifiers means that we do not offer any language support for (as an example) an immutable list of mutable objects.

In our approach uniqueness is guaranteed by linearity, that is, by allowing at most one use of an affine reference, rather than by destructive reads as in [13, 4].

Approaches based on destructive reads lead to the style of programming outlined below:

```
a.f=c.doStuff(a.f)//style suggested by others
//during execution of doStuff, 'a.f' is null
```

The object referenced by `a` has an *isolated* field `f` containing an object `b`. This object `b` is passed to a client `c`, which can use (potentially modifying) it. A typical pattern is that the result of such computation is a reference to `b`, which `a` can then recover. This approach allows *isolated* fields, as shown above, but has a serious drawback: an *isolated* field can become unexpectedly not available (in the example, during execution of `doStuff`), hence any object contract involving such field can be broken. In our approach, fields can not be affine: the “only once” use of capsule local variables, ensured by linear types, makes no sense on fields. Levereging on our sharing control, previous code can be rewritten as follows:

```
c.doStuff(a.f())//our suggested style
//doStuff guarantees absence of aliasing: A=empty
//during execution of doStuff, 'a.f()' is there
```

Alias analysis is a fundamental static analysis, used in compilers and code analysers. Algorithms such as Steensgaard’s algorithm, [25], infer equivalence classes that may alias. In [10] is presented a refined version of such algorithm, performing a uniqueness analysis, similar to our detection of “capsule” values. However, the aim of our work is to design a language in which annotations, such as the affine modifier, can be used by the user to enforce properties of its code. Then the inference system checks that such annotations are correctly used.

Finally, an important distinguishing feature of our work is that sharing can be directly represented at the syntactic level as a relation among free variables, thanks to the fact that the calculus is pure. Models of the imperative paradigm as pure calculi have been firstly proposed in [21, 5].

8 Conclusion

We have presented a type and effect system which *infers* sharing possibly introduced by the evaluation of an expression. As shown by the examples of Sect.4, this type system is very powerful. Notably, it discriminates between well-typed and ill-typed terms in situations where type systems based on declaring modifiers are either too restrictive or require rather tricky and non algorithmic rules [13, 24, 12]. Sharing is directly represented at the syntactic level as a relation among free variables, thanks to the fact that the calculus is *pure*. That is, imperative features are modeled by just rewriting source code terms.

In this operational semantics, reduction is defined on typechecked terms, where blocks have been annotated with the information of which local variables will be connected to the result. In this way, it is possible to define a rather sophisticated notion of congruence among terms, which allows to move declarations outside from a block only when this preserves well-typedness. Results include, besides soundness, the fact that the evaluation of an expression has a sharing effect only among those inferred by the type system.

In further work, we plan to enrich the type system to also handle *immutable* references. We will also investigate (a form of) Hoare logic on top of our 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, similarly to what is achieved by separation logic [20].

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 Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Fraction-polymorphic permission inference.

4 John Boyland. Semantics of fractional permissions with nesting. *ACM Transactions on Programming Languages and Systems*, 32(6), 2010.

5 Andrea Capriccioli, Marco Servetto, and Elena Zucca. An imperative pure calculus. *Electronic Notes in Theoretical Computer Science*, 322:87–102, 2016.

6 Elias Castegren and Tobias Wrigstad. Reference capabilities for concurrency control. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *ECOOP'16 - Object-Oriented Programming*, volume 56 of *LIPIcs*, pages 5:1–5:26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

7 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.

8 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.

9 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.

10 Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for java with strong updates. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 665–687. Springer, 2012. URL: `http://dx.doi.org/10.1007/978-3-642-31057-7_29`, `doi:10.1007/978-3-642-31057-7_29`.

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 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. URL: `http://ceur-ws.org/Vol-1720/full5.pdf`.

13 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.

14 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.

15 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.

16 Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 97–116. ACM, 2009. URL: `http://doi.acm.org/10.1145/1640089.1640097`, `doi:10.1145/1640089.1640097`.

17 John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journ. of Functional Programming*, 8(3):275–317, 1998.

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

19 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.

20 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.

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

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

23 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.

24 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.

25 Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In Tibor Gyimóthy, editor, *Compiler Construction, 6th International Conference, CC'96, Linköping, Sweden, April 24-26, 1996, Proceedings*, volume 1060 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 1996. URL: `http://dx.doi.org/10.1007/3-540-61053-7_58`, `doi:10.1007/3-540-61053-7_58`.

26 Aaron Turon. Rust: from POPL to practice (keynote). In Giuseppe Castagna and Andrew D. Gordon, editors, *ACM Symp. on Principles of Programming Languages 2017*, page 2. ACM Press, 2017.