Coeffects for Sharing and Mutation

RICCARDO BIANCHINI, Università di Genova, Italy
FRANCESCO DAGNINO, Università di Genova, Italy
PAOLA GIANNINI, Università del Piemonte Orientale, Italy
ELENA ZUCCA, Università di Genova, Italy
MARCO SERVETTO, Victoria University of Wellington, New Zealand

In *type-and-coeffect systems*, contexts are enriched by *coeffects* modeling how they are actually used, typically through annotations on single variables. Coeffects are computed bottom-up, combining, for each term, the coeffects of its subterms, through a fixed set of algebraic operators. We show that this principled approach can be adopted to track *sharing* in the imperative paradigm, that is, links among variables possibly introduced by the execution. This provides a significant example of non-structural coeffects, which cannot be computed by-variable, since the way a given variable is used can affect the coeffects of other variables. To illustrate the effectiveness of the approach, we enhance the type system tracking sharing to model a sophisticated set of features related to uniqueness and immutability. Thanks to the coeffect-based approach, we can express such features in a simple way and prove related properties with standard techniques.

CCS Concepts: • **Theory of computation → Program analysis**; **Type structures**.

Additional Key Words and Phrases: coeffect systems, sharing, Java

ACM Reference Format:
Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, and Marco Servetto. 2022. Coeffects for Sharing and Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 156 (October 2022), 29 pages. https://doi.org/10.1145/3563319

1 INTRODUCTION

Recently, *coeffect systems* have received considerable interest as a mechanism to reason about resource usage [Atkey 2018; Brunel et al. 2014; Choudhury et al. 2021; Dal Lago and Gavazzo 2022; Gaboardi et al. 2016; Ghica and Smith 2014; Orchard et al. 2019; Petricek et al. 2014]. They are, in a sense, the dual of effect systems: given a generic type judgment $\Gamma \vdash e : T$, effects can be seen as an enrichment of the type T (modeling side effects of the execution), whereas coeffects can be seen as an enrichment of the context Γ (modeling how execution needs to use external resources). In the typical case when Γ is a map from variables to types, the type judgment takes shape $x_1 :_{c_1} T_1, \ldots, x_n :_{c_n} T_n \vdash e : T$, where the *scalar* coeffect[1] c_i models how variable x_i is used in e. Such annotations on variables are an output of the typing process rather than an input: they are computed bottom-up, as a linear combination, for each term, of those of its subterms.

[1]Also called *grade*, using the terminology *graded type system rather than coeffect system.*

Authors' addresses: Riccardo Bianchini, DIBRIS, Università di Genova, Italy, riccardo.bianchini@edu.unige.it; Francesco Dagnino, francesco.dagnino@dibris.unige.it, DIBRIS, Università di Genova, Italy; Paola Giannini, DiSSTE, Università del Piemonte Orientale, Italy, paola.giannini@uniupo.it; Elena Zucca, elena.zucca@unige.it, DIBRIS, Università di Genova, Italy, elena.zucca@unige.it; Marco Servetto, ECS, Victoria University of Wellington, New Zealand, marco.servetto@vuw.ac.nz.

2475-1421/2022/10-ART156
https://doi.org/10.1145/3563319

The fact that a program introduces sharing between two variables, say x and y, for instance through a field assignment $x.f = y$ in an object-oriented language, clearly has a coeffect nature (indeed, it is a particular way to use the resources x and y). However, to the best of our knowledge no attempt has been made to use coeffects to track this information statically.

A likely reason is that this kind of coeffect does not fit in the framework of *structural* coeffect systems, which predominate in the literature, and where the coeffect of each single variable is computed independently. Clearly, this is not the case for sharing, since a program introducing sharing between x and y, and between y and z, introduces sharing between x and z as well.

In this paper, we show that sharing can be naturally modeled by coeffects following a more general schema distilling their fundamental ingredients; namely, a *semiring* for scalar coeffects, and a *module* structure for coeffect contexts, providing *sum* and *scalar multiplication*. Whereas scalars are regularly assumed in the literature to form (some variant of) a semiring, the fact that coeffect contexts form a module has only been noted, to the best of our knowledge, by McBride [2016] and subsequently by Wood and Atkey [2022]. In these papers, however, only structural coeffects are considered, that is, modules where sum and scalar multiplication are defined pointwise. Sharing coeffects provide a significant non-structural instance of the framework, motivating its generality.

To define the sharing coeffect system, we take as a reference language an imperative variant of Featherweight Java [Igarashi et al. 1999], and we extend the standard type system of the language by adding coeffects which track sharing introduced by the execution of a program. Following the guiding principle of coeffect systems, they are computed bottom up, starting from the rule for variables and constructing more complex coeffects by sums and scalar multiplications, where coefficients are determined by each language construct. Hence, the typing rules can be easily turned into an algorithm. In the resulting type-and-coeffect system, we are able to detect, in a simple and static way, some relevant notions in the literature, notably that an expression is a *capsule*[2], that is, evaluates to the *unique entry point* for a portion of memory.

To illustrate the effectiveness of this approach, we enhance the type system tracking sharing to model a sophisticated set of features related to uniqueness and immutability. Notably, we integrate and formalize the language designs proposed in [Giannini et al. 2019a,b], which have two common key ideas. The first is to use *modifiers* (read, caps, and `imm` for read-only, uniqueness, and immutability, respectively), allowing the programmer to specify the corresponding constraints/properties in variable/parameter declarations and method return types. The second is that uniqueness and immutability (caps and `imm` tags) are not *imposed*, but *detected* by the type system, supporting *ownership transfer* rather than *ownership invariants* (see Sect. 7).

Because it is built on top of the sharing coeffects, the type-and-coeffect system we design to formalize the above features significantly improves the previous work [Giannini et al. 2019a,b][3]. Notably, features from both papers are integrated; inference of caps and `imm` types is *straightforward* from the coeffects, through a simple *promotion* rule; the design of the type system is guided, and rules can be easily turned into an algorithm. Finally, the coeffect system uniformly computes both sharing introduced by a program *existing in current memory*, allowing us to express and prove relevant properties in a clean way and with standard techniques.

In summary, the contributions of the paper are the following:

- A schema distilling the ingredients of coeffect systems, mentioned by McBride [2016] and Wood and Atkey [2022], but never used in its generality, that is, beyond structural instances.

[2]We adopt the terminology of Giannini et al. [2019a,b]; in the literature there are many variants of this notion with different names [Almeida 1997; Clarke and Wrigstad 2003; Dietl et al. 2007; Gordon et al. 2012; Hogg 1991; Servetto et al. 2013].
[3]A more detailed comparison is provided in Sect. 8.

- The first, to the best of our knowledge, formalization of sharing by a coeffect system. We prove subject reduction, stating that not only is type preserved, but also sharing.
- On top of such a coeffect system, an enhanced type system supporting sophisticated features related to uniqueness and immutability. We prove subject reduction, stating that type, sharing, and modifiers are preserved, so that we can statically detect uniqueness and immutability.

We stress that the aim of the paper is *not* to propose a novel design of memory-management features, but to provide, via a complex example, a proof-of concept that coeffects can be the basis for modeling such features, which could be fruitfully employed in other cases. In particular, we demonstrate the following:

- The paradigm of coeffects can be applied for useful purposes in an imperative/OO setting, whereas so far in the literature it has only been used in functional calculi and languages.
- The views of ownership as a substructural capability and as a graph-theoretic property of heaps can be unified.
- The expressive power obtained in [Giannini et al. 2019a,b] by complicated and ad-hoc type systems is achieved in a much more elegant and principled way, using only simple algebraic operations. Moreover, the coeffect approach allows one to reuse existing general results regarding algorithms/implementations, like, e.g., those used in Granule [Orchard et al. 2019].

In Sect. 2 we present the reference language, and illustrate the properties we want to guarantee. In Sect. 3 we illustrate the ingredients of coeffect systems through a classical example, and we define their general algebraic structure. In Sect. 4 and Sect. 5 we describe the two type systems with the related results. In Sect. 6 we discuss the expressive power, compared with closely related proposals. In Sect. 7 we outline other related work, and in Sect. 8 we summarize our contribution and discuss future work. Omitted proofs can be found in [Bianchini et al. 2022b].

2 SHARING AND MUTATION IN A JAVA-LIKE CALCULUS

We illustrate the properties we want to guarantee with the coeffect system on a simple reference language, an imperative variant of Featherweight Java [Igarashi et al. 1999].

2.1 The Language

For the reader's convenience, syntax, reduction rules, and the standard type system are reported in Fig. 1. We write *es* as a metavariable for $e_1, \ldots, e_n$, $n \geq 0$, and analogously for other sequences. Expressions of primitive types, unspecified, include constants k. We assume a unique set of *variables* $x, y, z, \ldots$ which occur both in source code (method parameters, including the special variable `this`, and local variables in blocks) and as references in memory. Moreover, we assume sets of *class names* C, *field names* f, and *method names* m. In addition to the standard constructs of imperative object-oriented languages (field access, field assignment, and object creation), we have a block expression, consisting of a local variable declaration, and the body in which this variable can be used. We will sometimes abbreviate $\{T\ x = e;\ e'\}$ by $e;\ e'$ when x does not occur free in e.

To be concise, the class table is abstractly modeled as follows, omitting its (standard) syntax:

- $\mathsf{fields}(C)$ gives, for each class C, the sequence $T_1\ f_1; \ldots T_n\ f_n;$ of its fields with their types;
- $\mathsf{mbody}(C, m)$ gives, for each method m of class C, its parameters and body
- $\mathsf{mtype}(C, m)$ gives, for each method m of class C, its parameter types and return type.

For simplicity, we do not consider subtyping (inheritance), which is an orthogonal feature.

Method bodies are expected to be well-typed with respect to method types. Formally, $\mathsf{mbody}(C, m)$ and $\mathsf{mtype}(C, m)$ are either both defined or both undefined; in the first case $\mathsf{mbody}(C, m) = (x_1 \ldots x_n, e)$, $\mathsf{mtype}(C, m) = T_1 \ldots T_n \to T$, and

$\texttt{this} : \mathsf{C}, x_1 : T_1 \ldots, x_n : T_n \vdash e : T$

holds. Reduction is defined over *configurations* of shape $e|\mu$, where a *memory* μ is a map from references to *objects* of shape $[v_1, \ldots, v_n]^C$, and we assume free variables in e to be in $\mathsf{dom}(\mu)$. We denote by $\mu^{x.i=v}$ the memory obtained from μ by updating the i-th field of the object associated to x by v, and by $e[v/x]$ the usual capture-avoiding substitution.

Reduction and typing rules are straightforward. In rule (T-CONF), a configuration is well-typed if the expression is well-typed, and the memory is well-formed in the same context (recall that free variables in the expression are bound in the domain of the memory). In rule (T-MEM), a memory is well-formed in a context assigning a type to all and only references in memory, provided that, for each reference, the associated object has the same type.

2.2 Sharing and Mutation

In languages with state and mutations, keeping control of sharing is a key issue for correctness. This is exacerbated by concurrency mechanisms, since side-effects in one thread can affect the behaviour of another, hence unpredicted sharing can induce unplanned/unsafe communication.

Sharing means that some portion of the memory can be reached through more than one reference, say through x and y, so that manipulating the memory through x can affect y as well.

Definition 2.1 (Sharing in memory). The *sharing relation* in memory μ, denoted by $\bowtie_\mu$, is the smallest equivalence relation on $\mathsf{dom}(\mu)$ such that:

$x \bowtie_\mu y$ if $\mu(x) = [v_1, \ldots, v_n]^C$ and $y = v_i$ for some $i \in 1..n$

Note that $y = v_i$ above means that y and v_i *are the same reference*, that is, it corresponds to what is sometimes called pointer equality.

It is important for a programmer to be able to rely on *capsule* and *immutability* properties. Informally, an expression has the capsule property if its result will be the *unique entry point* for a portion of store. For instance, we expect the result of a `clone` method to be a capsule, see Example 2.5 below. This allows programmers to identify state that can be safely used by a thread since no other thread can access/modify it. A reference has the immutability property if its reachable object graph will be never modified. As a consequence, an immutable reference can be safely shared by threads.

The following simple example illustrates the capsule property.

Example 2.2. Assume the following class table:

```
class B {int f;}
class C {B f1; B f2;}
```

and consider the expression $e = \{\mathsf{B}\,\mathsf{z} = \texttt{new B(2); x.f1=y; new C(z,z)}\}$. This expression has two free variables (in other words, uses two external resources) x and y. We expect such free variables to be bound to an outer declaration, if the expression occurs as a subterm of a program, or to represent references in current memory. This expression is a *capsule*. Indeed, even though it has free variables (uses external resources) `x` and `y`, such variables will *not* be connected to the final result. We say that they are *lent* in e. In other words, lent references can be manipulated during the evaluation, but cannot be permanently saved. So, we have the guarantee that the result of evaluating e, regardless of the initial memory, will be a reference pointing to a *fresh* portion of memory. For instance, evaluating e in $\mu = \{\mathsf{x} \mapsto [\mathsf{x1}, \mathsf{x1}]^\mathsf{C}, \mathsf{x1} \mapsto [0]^\mathsf{B}, \mathsf{y} \mapsto [1]^\mathsf{B}\}$, the final result is a fresh reference `w`, in the memory $\mu' = \{\mathsf{x} \mapsto [\mathsf{y}, \mathsf{x1}]^\mathsf{C}, \mathsf{x1} \mapsto [0]^\mathsf{B}, \mathsf{y} \mapsto [1]^\mathsf{B}, \mathsf{z} \mapsto [2]^\mathsf{B}, \mathsf{w} \mapsto [\mathsf{z}, \mathsf{z}]^\mathsf{C}\}$.

Lent and capsule properties are formally defined below.

Definition 2.3 (Lent reference). For $x \in \mathsf{fv}(e)$, x is lent in e if, for all μ, $e|\mu \rightarrow^\star y|\mu'$ implies $x \bowtie_{\mu'} y$ does not hold.

$$
\begin{array}{lcll}
e & ::= & x \mid k \mid e.f \mid e.f = e' \mid \mathsf{new}\ C(es) \mid e.m(es) \mid \{T\,x = e;\ e'\} \mid \ldots & \text{expression} \\
T & ::= & C \mid P & \text{type} \\
\\
v & ::= & x \mid k & \text{value} \\
\mathcal{E} & ::= & [\,] \mid \mathcal{E}.f \mid \mathcal{E}.f = e' \mid x.f = \mathcal{E} \mid \mathsf{new}\ C(vs, \mathcal{E}, es) & \text{evaluation context} \\
 & & \mid \mathcal{E}.m(es) \mid x.m(vs, \mathcal{E}, es) \mid \{T\,x = \mathcal{E};\ e\} \mid \ldots &
\end{array}
$$

$$
\textsc{(ctx)}\ \frac{e|\mu \to e'|\mu'}{\mathcal{E}[e]|\mu \to \mathcal{E}[e']|\mu'} \qquad \textsc{(field-access)}\ \frac{\begin{array}{l}\mu(x) = \mathsf{new}\ C(v_1, \ldots, v_n) \\ \mathsf{fields}(C) = T_1\,f_1; \ldots T_n\,f_n; \\ i \in 1..n\end{array}}{x.f_i|\mu \to v_i|\mu}
$$

$$
\textsc{(field-assign)}\ \frac{\begin{array}{l}\mu(x) = [v_1, \ldots, v_n]^C \\ \mathsf{fields}(C) = T_1\,f_1; \ldots T_n\,f_n; \\ i \in 1..n\end{array}}{x.f_i = v|\mu \to v|\mu^{x.i=v}}
$$

$$
\textsc{(new)}\ \frac{}{\mathsf{new}\ C(vs)|\mu \to x|\mu[\mathsf{new}\ C(vs)/x]}\ x \notin \mathsf{dom}(\mu)
$$

$$
\textsc{(invk)}\ \frac{}{x.m(v_1, \ldots, v_n)|\mu \to e[x/\mathtt{this}][v_1/x_1]\ldots[v_n/x_n]|\mu}\ \begin{array}{l}\mu(x) = \mathsf{new}\ C(vs) \\ \mathsf{mbody}(C, m) = (x_1 \ldots x_n, e)\end{array}
$$

$$
\textsc{(block)}\ \frac{}{\{T\,x = v;\ e\}|\mu \to e[v/x]|\mu}
$$

$$
\textsc{(t-var)}\ \frac{}{\Gamma \vdash x : T}\ \Gamma(x) = T \qquad \textsc{(t-const)}\ \frac{}{\Gamma \vdash k : P_k}
$$

$$
\textsc{(t-field-access)}\ \frac{\Gamma \vdash e : C}{\Gamma \vdash e.f_i : T_i}\ \begin{array}{l}\mathsf{fields}(C) = T_1\,f_1; \ldots T_n\,f_n; \\ i \in 1..n\end{array}
$$

$$
\textsc{(t-field-assign)}\ \frac{\Gamma \vdash e : C \quad \Gamma \vdash e' : T_i}{\Gamma \vdash e.f_i = e' : T_i}\ \begin{array}{l}\mathsf{fields}(C) = T_1\,f_1; \ldots T_n\,f_n; \\ i \in 1..n\end{array}
$$

$$
\textsc{(t-new)}\ \frac{\Gamma \vdash e_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash \mathsf{new}\ C(e_1, \ldots, e_n) : C}\ \mathsf{fields}(C) = T_1\,f_1; \ldots T_n\,f_n;
$$

$$
\textsc{(t-invk)}\ \frac{\Gamma \vdash e_0 : C \qquad \Gamma \vdash e_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash e_0.m(e_1, \ldots, e_n) : T}\ \mathsf{mtype}(C, m) = T_1 \ldots T_n \to T
$$

$$
\textsc{(t-block)}\ \frac{\Gamma \vdash e : T \qquad \Gamma, x : T \vdash e' : T'}{\Gamma \vdash \{T\,x = e;\ e'\} : T'}
$$

$$
\textsc{(t-conf)}\ \frac{\Gamma \vdash e : T \qquad \Gamma \vdash \mu}{\Gamma \vdash e|\mu : T} \qquad \textsc{(t-obj)}\ \frac{\Gamma \vdash v_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash [v_1, \ldots, v_n]^C : C}\ \mathsf{fields}(C) = T_1\,f_1; \ldots T_n\,f_n;
$$

$$
\textsc{(t-mem)}\ \frac{\Gamma \vdash \mu(x_i) : C_i \quad \forall i \in 1...n}{\Gamma \vdash \mu}\ \begin{array}{l}\Gamma = x_1 : C_1, \ldots, x_n : C_n \\ \mathsf{dom}(\Gamma) = \mathsf{dom}(\mu)\end{array}
$$

Fig. 1. Syntax, reduction rules, and standard type system of the Java-like calculus

An expression e is a capsule if all its free variables are lent in e.

Definition 2.4 (Capsule expression). An expression e is a *capsule* if, for all μ, $e|\mu \rightarrow^\star y|\mu'$ implies that, for all $x \in \mathsf{fv}(e)$, $x \bowtie_{\mu'} y$ does not hold.

The capsule property can be easily detected in simple situations, such as using a primitive deep clone operator, or a closed expression. However, the property also holds in many other cases, which are *not* easily detected (statically) since they depend on *the way variables are used*. To see this, we consider a more involved example, adapted from [Giannini et al. 2019b].

Example 2.5.

```
class B {int f; B clone() {new B(this.f)}
class A { B f;
  A mix (A a) {this.f=a.f; a} // this, a and result linked
  A clone () {new A(this.f.clone())} // this and result not linked
}
A a1 = new A(new B(0));
A mycaps = {A a2 = new A(new B(1));
  a1.mix(a2).clone() // (1)
  // a1.mix(a2).clone().mix(a2) // (2)
}
```

The result of `mix`, as the name suggests, will be connected to both the receiver and the argument, whereas the result of `clone`, as expected for such a method, will be a reference to a *fresh* portion of memory which is not connected to the receiver.

Now let us consider the code after the class definition, where the programmer wants the guarantee that `mycaps` will be initialized with a capsule, that is, an expression which evaluates to the entry point of a fresh portion of memory. Fig. 2 shows a graphical representation of the store after the

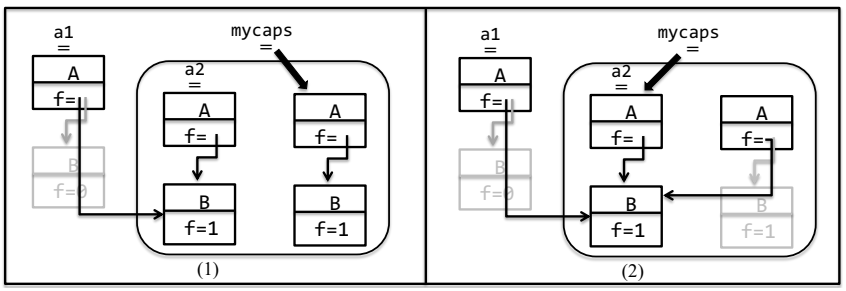

Fig. 2. Graphical representation of the store for Example 2.5

evaluation of such code. Side (1) shows the resulting store if we evaluate line (1) but not line (2), while side (2) shows the resulting store if we evaluate line (2) but not line (1). The thick arrow points to the result of the evaluation of the block and `a2` is a local variable. In side (1) `a1` is not in sharing with `mycaps`, whereas in side (2) `a1` is in sharing with `a2` which is in sharing with `mycaps` and so `a1` is in sharing with `mycaps` as well. Set e_1 = `{A a2 = new A(new B(1)); a1.mix(a2).clone()}` and e_2 = `{A a2 = new A(new B(1)); a1.mix(a2).clone().mix(a2)}`. We can see that `a1` is lent in e_1, since its evaluation produces the object pointed to by the thick arrow which is not in sharing with `a1`, whereas `a1` is not lent in e_2. Hence, e_1 is a capsule, since its free variable, `a1`, is not in sharing with the result of its evaluation, whereas `a2` is not.

We consider now immutability. A reference x has the immutability property if the portion of memory reachable from x will never change during execution, as formally stated below.

Definition 2.6. The *reachability relation* in memory μ, denoted by $\triangleright_\mu$, is the reflexive and transitive closure of the relation on $\mathrm{dom}(\mu)$ such that:

$$x \triangleright_\mu y \text{ if } \mu(x) = [v_1, \ldots, v_n]^C \text{ and } y = v_i \text{ for some } i \in 1..n$$

Definition 2.7 (Immutable reference). For $x \in \mathrm{fv}(e)$, x is *immutable* in e if $e|\mu \rightarrow^\star e'|\mu'$ and $x \triangleright_\mu y$ implies $\mu(y) = \mu'(y)$.

A typical way to prevent mutation, as we will show in Sect. 5, is by a type modifier read, so that an expression with type tagged in this way cannot occur as the left-hand side of a field assignment. However, to have the guarantee that a certain portion of memory is actually immutable, a type system should be able to detect that it cannot be modified through *any* possibile reference. For instance, consider a variant of Example 2.5 with the same classes `A` and `B`.

Example 2.8.

```
A a1 = new A(new B(0));
read A mycaps = {A a2 = new A(new B(1));
  a1.mix(a2).clone() // (1)
  // a1.mix(a2).clone().mix(a2) // (2)
}
// mycaps.f.f=3 // (3)
a1.f.f=3 // (4)
```

The reference `mycaps` is now declared as a read type, hence we cannot modify its reachable object graph through `mycaps`. For instance, line (3) is ill-typed. However, if we replace line (1) with line (2), since in this case `mycaps` and `a1` share their `f` field, the same effect of line (3) can be obtained by line (4). This example shows that the immutability property is, roughly, a conjunction of the read restriction and the capsule property.

3 COEFFECT SYSTEMS

In Sect. 3.1 we illustrate the fundamental ingredients of coeffect systems through a classical example, and in Sect. 3.2 we formally define their general algebraic structure.

3.1 An Example

In Fig. 3 we show the example which is generally used to illustrate how a coeffect system works[4]. Namely, a simple coeffect system for the call-by-name λ-calculus where we trace when a variable is either not used, or used linearly (that is, exactly once), or used in an unrestricted way, as expressed by assigning to the variable a *scalar coeffect* c.

A *coeffect context*, of shape $\gamma = x_1 : c_1, \ldots, x_n : c_n$, where order is immaterial and $x_i \neq x_j$ for $i \neq j$, represents a map from variables to scalar coeffects where only a finite number of variables have non-zero coeffect. A *(type-and-coeffect) context*, of shape $\Gamma = x_1 :_{c_1} T_1, \ldots, x_n :_{c_n} T_n$, with analogous conventions, represents the pair of the standard type context $x_1 : T_1 \ldots, x_n : T_n$, and the coeffect context $x_1 : c_1, \ldots, x_n : c_n$. We write $\mathrm{dom}(\Gamma)$ for $\{x_1, \ldots, x_n\}$.

Scalar coeffects usually form a preordered semiring [Abel and Bernardy 2020; Atkey 2018; Brunel et al. 2014; Choudhury et al. 2021; Gaboardi et al. 2016; Ghica and Smith 2014; McBride 2016;

[4]More precisely, the structure of coeffects is that of most papers cited in the Introduction, and the calculus a variant/combination of examples in those papers.

$$
\begin{array}{lcl}
t & ::= & n \mid x \mid \lambda x{:}T.t \mid t_1\ t_2 \\
c & ::= & 0 \mid 1 \mid \omega \\
T & ::= & \mathsf{int} \mid T_1 \xrightarrow{c} T_2 \\
\gamma & ::= & x_1 : c_1, \ldots, x_n : c_n \\
\Gamma, \Delta & ::= & x_1 :_{c_1} T_1, \ldots, x_n :_{c_n} T_n
\end{array}
\qquad (\textsc{AppAbs})\ \frac{}{(\lambda x{:}T.t)\ t' \to t[t'/x]} \qquad (\textsc{App})\ \frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}
$$

$$
(\textsc{t-const})\ \frac{}{\emptyset \vdash n : \mathsf{int}} \qquad (\textsc{t-var})\ \frac{}{0 \times \Gamma + x :_1 T \vdash x : T} \qquad (\textsc{t-sub})\ \frac{\Gamma \vdash t : T}{\Gamma' \vdash t : T}\ \Gamma' \preceq \Gamma
$$

$$
(\textsc{t-abs})\ \frac{\Gamma, x :_c T_1 \vdash t : T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \xrightarrow{c} T_2} \qquad (\textsc{t-app})\ \frac{\Gamma_1 \vdash t_1 : T_2 \xrightarrow{c} T_1 \qquad \Gamma_2 \vdash t_2 : T_2}{\Gamma_1 + ((c \vee 1) \times \Gamma_2) \vdash t_1\ t_2 : T_1}
$$

Fig. 3. A simple structural coeffect system

Orchard et al. 2019; Wood and Atkey 2022], that is, they are equipped with a *preorder* $\preceq$ (with binary join $\vee$), a *sum* $+$, and a *multiplication* $\times$, satisfying some natural axioms, see Def. 3.1 in Sect. 3.2. In the example, the (pretty intuitive) definition of such a structure is given below.

$0 \preceq \omega,\ 1 \preceq \omega$

$+$	0	1	ω
0	0	1	ω
1	1	ω	ω
ω	ω	ω	ω

$\times$	0	1	ω
0	0	0	0
1	0	1	ω
ω	0	ω	ω

The typing rules use three operators on contexts: *preorder* $\preceq$, *sum* $+$ and *multiplication* $\times$ of a scalar coeffect with a context. In the example, these operators are defined by first taking, on coeffect contexts, the pointwise application of the corresponding scalar operator, with the warning that the inverse preorder on scalars is used, see rule (T-SUB) below. Then, they are lifted to type-and-coeffect contexts, resulting in the following definitions:

- $\Gamma \preceq \Delta$ is the preorder defined by
  $(0 \times \Delta), \Gamma \preceq \Gamma \qquad (x :_c T, \Gamma) \preceq (x :_{c'} T, \Delta)$ if $c' \preceq c$ and $\Gamma \preceq \Delta$
- $c \times \Gamma$ is the context defined by
  $c \times \emptyset = \emptyset \qquad c \times (x :_{c'} T, \Gamma) = x :_{c \times c'} T, (c \times \Gamma)$
- $\Gamma + \Delta$ is the context defined by
  $\emptyset + \Gamma = \Gamma \qquad (x :_c T, \Gamma) + \Delta = x :_c T, (\Gamma + \Delta)$ if $x \notin \mathsf{dom}(\Delta)$
  $(x :_c T, \Gamma) + (x :_{c'} T, \Delta) = x :_{c+c'} T, (\Gamma + \Delta)$

Note that when lifted to type-and-coeffect contexts the sum becomes partial, since we require a common variable to have the same type.

In rule (T-CONST) no variable is used. In rule (T-VAR), the coeffect context is one of those representing the map where the given variable is used exactly once, and no other is used. Indeed, $0 \times \Gamma$ is a context where all variables have 0 coeffect. We include, to show the role of $\preceq$, a standard subsumption rule (T-SUB), allowing a well-typed expression to be typed in a more specific context, where coeffects are overapproximated.[5] This rule becomes useful, e.g., in the presence of a conditional construct, as its typing rule usually requires the two branches to be typed in the same context (that is, to use resources in the same way) and subsumption relaxes this condition. In rule (T-ABS), the type of a lambda expression is decorated with the coeffect assigned to the binder when typechecking the body. In rule (T-APP), the coeffects of an application are obtained by summing the coeffects of the first subterm, which is expected to have a functional type decorated with a coeffect, and the

[5]Note that this rule partly overlaps with (T-VAR).

coeffects of the argument multiplied by the decoration of the functional type. The part emphasized in gray, which shows the use of the join operator, needs to be added in a call-by-value strategy. For instance, without this addition, the judgment $y :_0 \mathsf{int} \vdash (\lambda x{:}\mathsf{int}.n)\ y : \mathsf{int}$ holds, meaning that y is not actually used, whereas it is used in call-by-value.

Extrapolating from the example, we can distill the following ingredients of a coeffect system:

- The typing rules use three operators on contexts (preorder, sum, and scalar multiplication) defined on top of the corresponding scalar operators.
- Coeffects are computed bottom-up, starting from the rule for variable.
- As exemplified in (T-APP), the coeffects of a compound term are computed by a *linear combination* (through sum and scalar multiplication) of those of the subterms. The coefficients are determined by the specific language construct considered in the typing rule.
- The preorder is used for overapproximation.

Note also that, by just changing the semiring of scalars, we obtain a different coeffect system. For instance, an easy variant is to consider the natural numbers (with the usual preorder, sum, and multiplication) as scalar coeffects, tracking *exactly how many times* a variable is used. The definition of contexts and their operations, and the typing rules, can be kept exactly the same.

In the following section, we will provide a formal account of the ingredients described above.

3.2 The Algebra of Coeffects

As illustrated in the previous section, the first ingredient is a *(preordered) semiring*, whose elements abstract a "measure" of resource usage.

Definition 3.1 (Semiring). A *(preordered) semiring* is a tuple $\mathcal{R} = \langle R, \preceq, +, \times, 0, 1\rangle$ where

- $\langle R, \preceq\rangle$ is a preordered set
- $\langle R, +, 0\rangle$ is an ordered commutative monoid
- $\langle R, \times, 1\rangle$ is an ordered monoid

such that the following equalities hold for all $r, s, t \in R$

$$(r + s) \times t = (r \times t) + (s \times t) \qquad r \times (s + t) = (r \times s) + (r \times t)$$
$$r \times 0 = 0 \qquad 0 \times r = 0$$

Spelling out the definition, this means that both $+$ and $\times$ are associative and monotone with respect to $\preceq$, and $+$ is also commutative. In the following we will adopt the usual precedence rules for addition and multiplication.

Let us assume a semiring $\mathcal{R} = \langle R, \preceq, +, \times, 0, 1\rangle$ throughout this section. Again, as exemplified in the previous section, coeffect contexts have a preorder, a sum with a neutral element and a multiplication by elements of the semiring. Formally, they form a *module over the semiring*, as already observed by McBride [2016] and Wood and Atkey [2022].

Definition 3.2 ($\mathcal{R}$-module). A *(preordered) $\mathcal{R}$-module* $\mathcal{M}$ is a tuple $\langle M, \preceq, +, \mathbf{0}, \times\rangle$ where

- $\langle M, \preceq\rangle$ is a preordered set
- $\langle M, +, \mathbf{0}\rangle$ is a commutative monoid
- $\times\colon R{\times}M \to M$ is a function, called *scalar multiplication*, which is monotone in both arguments and satisfies the following equalities:

$$(r + s) \times u = (r \times u) + (s \times u) \qquad r \times (u + v) = (r \times u) + (r \times v) \qquad (r \times s) \times u = r \times (s \times u)$$
$$0 \times u = \mathbf{0} \qquad r \times \mathbf{0} = \mathbf{0} \qquad 1 \times u = u$$

Given $\mathcal{R}$-modules $\mathcal{M}$ and $\mathcal{N}$, a *(lax) homomorphism* $f\colon \mathcal{M} \to \mathcal{N}$ is a monotone function $f\colon M \to N$ such that the following hold for all $u, v \in M$ and $r \in R$:

$$f(u) + f(v) \preceq f(u + v) \qquad f(r \times u) = r \times f(u)$$

From the second equality it follows that $\mathbf{0} = f(\mathbf{0})$ as $\mathbf{0} = 0 \times f(\mathbf{0}) = f(0 \times \mathbf{0}) = f(\mathbf{0})$. It is also easy to see that $\mathcal{R}$-modules and their homomorphisms form a category, denoted by $\mathcal{R}$-Mod. Note that Wood and Atkey [2022] use a different notion of homomorphism built on relations. Here we prefered to stick to a more standard functional notion of homomorphism. The comparison between these two notions is an interesting topic for future work.

We show that the coeffects of the example in the previous section, and in general any structural coeffects, form an $\mathcal{R}$-module.

Let X be a set and $\alpha\colon X \to R$ be a function. The *support* of α is the set $\mathrm{supp}(\alpha) = \{x \in X \mid \alpha(x) \neq 0\}$. Denote by R^X the set of functions $\alpha\colon X \to R$ with finite support, then we can define the $\mathcal{R}$-module $\mathcal{R}^X = \langle R^X, \hat{\preceq}, \hat{+}, \hat{0}, \hat{\times}\rangle$ where $\hat{\preceq}$ and $\hat{+}$ are the pointwise extension of $\preceq$ and $+$ to R^X, $\hat{0}$ is the constant function equal to 0 and $r\hat{\times}\alpha = x \mapsto r \times \alpha(x)$, for all $r \in R$ and $\alpha \in R^X$. Note that $\hat{0}$, $\hat{+}$ and $\hat{\times}$ are well-defined because $\mathrm{supp}(\hat{0}) = \emptyset$, $\mathrm{supp}(\alpha\hat{+}\beta) \subseteq \mathrm{supp}(\alpha) \cup \mathrm{supp}(\beta)$ and $\mathrm{supp}(r\hat{\times}\alpha) \subseteq \mathrm{supp}(\alpha)$. When X is the set of variables, $\mathcal{R}^X$ (with the inverse preorder) is precisely the module of coeffect contexts in the structural case: they assign to each variable an element of the semiring and the requirement of finite support ensures that only finitely many variables have non-zero coeffect.

Finally, the coeffect systems considered in this paper additionally assume that the preordered semiring, hence the associated module, has binary joins. Since this assumption is completely orthogonal to the development in this section, we have omitted it. However, all definitions and results also work in presence of binary joins, hence they can be added without issues.

4 COEFFECTS FOR SHARING

Introducing sharing, e.g. by a field assignment $x.f = y$, can be clearly seen as adding an arc between x and y in an undirected graph where nodes are variables. However, such a graphical representation would be a *global* one, whereas the representation we are looking for must be *per variable*, and, moreover, must support sum and scalar multiplication operators. To achieve this, we introduce auxiliary entities called *links*, and attach to each variable a set of them, so that an arc between x and y is represented by the fact that they have a common link.[6] Moreover, there is a special link res which denotes a connection with the final result of the expression.

For instance, considering again the classes of Example 2.2:

```
class B {int f;}
class C {B f1; B f2;}
```

and the program `x.f1=y; new C(z1,z2)`, the following typing judgment will be derivable:

$$(*)\ \mathtt{x} :_{\{\ell\}} \mathtt{C}, \mathtt{y} :_{\{\ell\}} \mathtt{B}, \mathtt{z1} :_{\{\mathrm{res}\}} \mathtt{B}, \mathtt{z2} :_{\{\mathrm{res}\}} \mathtt{B} \vdash \mathtt{x.f1=y; new\ C(z1,z2)} : \mathtt{C} \quad \text{with } \ell \neq \mathrm{res}$$

meaning that the program's execution introduces sharing between x and y, as expressed by their common link ℓ, and between z1, z2, and the final result, as expressed by their common link res. The derivation for this judgment is shown later (Fig. 5).

Formally, we assume a countable set Lnk, ranged over by ℓ, with a distinguished element res. In the coeffect system for sharing, scalar coeffects X, Y, and Z will be finite sets of links. Let L be the finite powerset of Lnk, that is, the set of scalar coeffects, and let CCtx^L be the set of the corresponding coeffect contexts γ, that is (representations of) maps in L^V, with V the set of variables. Given $\gamma = x_1 : X_1, \ldots, x_n : X_n$, the *(transitive) closure* of γ, denoted $\gamma^\star$, is $x_1 : X_1^\star, \ldots, x_n : X_n^\star$ where $X_1^\star, \ldots, X_n^\star$ are the smallest sets such that:

[6] This roughly corresponds to the well-known representation of a (hyper)graph by a bipartite graph.

$\ell \in X_i$ implies $\ell \in X_i^\star$
$\ell, \ell' \in X_i^\star$, $\ell' \in X_j^\star$ implies $\ell \in X_j^\star$

For instance, if $\gamma = x : \{\ell\}, y : \{\ell, \ell'\}, z : \{\ell'\}$, then $\gamma^\star = x : \{\ell, \ell'\}, y : \{\ell, \ell'\}, z : \{\ell, \ell'\}$. That is, since x and y are connected by ℓ, and y and z are connected by ℓ', then x and z are connected as well. Note that, if γ is *closed* ($\gamma^\star = \gamma$), then two variables have either the same, or disjoint coeffects.

To sum two closed coeffect contexts, obtaining in turn a closed one, we need to apply the transitive closure after pointwise union. For instance, the above coeffect context γ could have been obtained as pointwise union of $x : \{\ell\}, y : \{\ell\}$ and $y : \{\ell'\}, z : \{\ell'\}$.

Multiplication of a closed coeffect context with a scalar is defined in terms of an operator $\triangleleft$ on sharing coeffects, which replaces the res link (if any) in the second argument with the first:

$$X \triangleleft Y = \begin{cases} \emptyset & \text{if } X = \emptyset \\ Y & \text{if } X \neq \emptyset \text{ and } \mathsf{res} \notin Y \\ (Y \setminus \{\mathsf{res}\}) \cup X & \text{if } X \neq \emptyset \text{ and } \mathsf{res} \in Y \end{cases}$$

Similarly to sum, to multiply a coeffect context with a scalar X, we need to apply the transitive closure after pointwise application of the operation $\triangleleft$. For instance, $\{\ell''\} \times (x : \{\ell, \mathsf{res}\}, y : \{\ell'\}) = x : \{\ell, \ell''\}, y : \{\ell'\}$. To see that transitive closure can be necessary, consider, for instance, $\{\ell''\} \times (x : \{\ell, \mathsf{res}\}, y : \{\ell''\}) = x : \{\ell, \ell''\}, y : \{\ell, \ell''\}$.

When an expression e, typechecked with context Γ, replaces a variable with coeffect X in an expression e', the product $X \times \Gamma$ computes the sharing introduced by the resulting expression on the variables in Γ. For instance, set $e = \texttt{x.f1=y; new C(z1, z2)}$ of $(*)$ and assume that e replaces `z` in `z.f1=w`, for which the judgment $\texttt{z} :_{\{\mathsf{res}\}} \texttt{C}, \texttt{w} :_{\{\mathsf{res}\}} \texttt{B} \vdash \texttt{z.f1=w} : \texttt{B}$ is derivable. We expect that `z1` and `z2`, being connected to the result of e, are connected to whatever `z` is connected to (`w` and the result of `z.f1=w`), whereas the sharing of `x` and `y` would not be changed. In our example, we have $\{\mathsf{res}\} \triangleleft \{\ell\} = \{\ell\}$ and $\{\mathsf{res}\} \triangleleft \{\mathsf{res}\} = \{\mathsf{res}\}$. Altogether we have the following formal definition:

Definition 4.1. The *sharing coeffect system* is defined by:

- the semiring $\mathcal{L} = (L, \subseteq, \cup, \triangleleft, \emptyset, \{\mathsf{res}\})$
- the $\mathcal{L}$-module $\langle \mathrm{CCtx}_\star^L, \hat{\subseteq}, +, \emptyset, \times \rangle$ where:
  - $\mathrm{CCtx}_\star^L$ are the fixpoints of $\star$, that is, the closed coeffect contexts
  - $\hat{\subseteq}$ is the pointwise extension of $\subseteq$ to $\mathrm{CCtx}_\star^L$
  - $\Gamma + \Gamma' = (\Gamma \hat{\cup} \Gamma')^\star$, where $\hat{\cup}$ is the pointwise extension of $\cup$ to $\mathrm{CCtx}_\star^L$
  - $X \times \Gamma = (X \hat{\triangleleft} \Gamma)^\star$, where $\hat{\triangleleft}$ is the pointwise extension of $\triangleleft$ to $\mathrm{CCtx}_\star^L$.

Operations on closed coeffect contexts can be lifted to type-and-coeffect contexts, exactly as we did in the introductory example in Sect. 3.1.

It is easy to check that $\mathcal{L} = (L, \subseteq, \cup, \triangleleft, \emptyset, \{\mathsf{res}\})$ is actually a semiring with $\emptyset$ neutral element of $\cup$ and $\{\mathsf{res}\}$ neutral element of $\triangleleft$. The fact that $\langle \mathrm{CCtx}_\star^L, \hat{\subseteq}, +, \emptyset, \times \rangle$ is actually an $\mathcal{L}$-module can be proved as follows: first of all, $\mathcal{L}^V = \langle \mathrm{CCtx}^L, \hat{\subseteq}, \hat{\cup}, \emptyset, \hat{\triangleleft} \rangle$ is an $\mathcal{L}$-module, notably, the structural one (all operations are pointwise); it is easy to see that $\_^\star$ is an idempotent homomorphism on $\mathcal{L}^V$; then, the thesis follows from a general result proved in [Bianchini et al. 2022b], stating that an idempotent homomorphism on a module induces a module on the set of its fixpoints.

In a judgment $\Gamma \vdash e : T$, the coeffects in Γ describe an equivalence relation on $\mathrm{dom}(\Gamma) \cup \{\mathsf{res}\}$ where each coeffect corresponds to an equivalence class. Two variables, say x and y, have the same coeffect if the evaluation of e possibly introduces sharing between x and y. Moreover, res in the coeffect of x models possible sharing with the final result of e. Intuitively, sharing only happens among variables of reference types (classes), since a variable x of a primitive type P denotes an

$$\begin{array}{lcll} \Gamma, \Delta & ::= & x_1 :_{X_1} T_1, \ldots, x_n :_{X_n} T_n & \text{context} \\ X & ::= & \{\ell_1, \ldots, \ell_n\} & \text{coeffect (set of links)} \end{array}$$

(T-VAR) $$\frac{}{\emptyset \times \Gamma + x :_{\{\mathsf{res}\}} T \vdash x : T}$$ (T-CONST) $$\frac{}{\emptyset \times \Gamma \vdash k : P_k}$$

(T-FIELD-ACCESS) $$\frac{\Gamma \vdash e : C}{\Gamma \vdash e.f_i : T_i} \quad \mathsf{fields}(C) = T_1\, f_1; \ldots T_n\, f_n; \quad i \in 1..n$$

(T-FIELD-ASSIGN) $$\frac{\Gamma \vdash e : C \quad \Delta \vdash e' : T_i}{\Gamma + \Delta \vdash e.f_i = e' : T_i} \quad \mathsf{fields}(C) = T_1\, f_1; \ldots T_n\, f_n; \quad i \in 1..n$$

(T-NEW) $$\frac{\Gamma_i \vdash e_i : T_i \quad \forall i \in 1..n}{\Gamma_1 + \ldots + \Gamma_n \vdash \mathsf{new}\, C(e_1, \ldots, e_n) : C} \quad \mathsf{fields}(C) = T_1\, f_1; \ldots T_n\, f_n;$$

(T-INVK) $$\frac{\Gamma_0 \vdash e_0 : C \qquad \Gamma_i \vdash e_i : T_i \quad \forall i \in 1..n}{\sum_{i=0}^{n}(X_i \cup \{\ell_i\}) \times \Gamma_i) \vdash e_0.m(e_1, \ldots, e_n) : T} \quad \mathsf{mtype}(C, m) \equiv^{\mathsf{fr}} X_0, T_1^{X_1} \ldots T_n^{X_n} \to T \quad \ell_0, \ldots, \ell_n \text{ fresh}$$

(T-BLOCK) $$\frac{\Gamma \vdash e : T \qquad \Gamma', x :_X T \vdash e' : T'}{(X \cup \{\ell\}) \times \Gamma + \Gamma' \vdash \{T\, x = e;\, e'\} : T'} \quad \ell \text{ fresh}$$ (T-PRIM) $$\frac{\Gamma \vdash e : P}{\{\ell\} \times \Gamma \vdash e : P} \quad \ell \text{ fresh}$$

(T-CONF) $$\frac{\Delta \vdash e : T \qquad \Gamma \vdash \mu}{\Delta + \Gamma \vdash e|\mu : T} \quad \mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\Gamma)$$

(T-OBJ) $$\frac{\Gamma_i \vdash v_i : T_i \quad \forall i \in 1..n}{\Gamma_1 + \cdots + \Gamma_n \vdash [v_1, \ldots, v_n]^C : C} \quad \mathsf{fields}(C) = T_1\, f_1; \ldots T_n\, f_n;$$

(T-MEM) $$\frac{\Gamma_i \vdash \mu(x_i) : C_i \quad \forall i \in 1...n}{\Gamma_\mu + \Gamma \vdash \mu} \quad \begin{array}{l} \Gamma_\mu = x_1 :_{\{\ell_1\}} C_1, \ldots, x_n :_{\{\ell_n\}} C_n \\ \mathsf{dom}(\Gamma_\mu) = \mathsf{dom}(\mu) \\ \Gamma = (\{\ell_1\} \times \Gamma_1) + \ldots + (\{\ell_n\} \times \Gamma_n) \\ \ell_1, \ldots, \ell_n \text{ fresh} \end{array}$$

Fig. 4. Coeffect system for sharing

immutable value rather than a reference in memory. To have a uniform treatment, a judgment $x :_{\{\ell\}} P \vdash x : P$ with ℓ fresh is derivable (by rules (T-VAR) and (T-PRIM), as detailed below[7]).

The typing rules are given in Fig. 4. In the rule for variable, the variable is obviously linked with the result (they coincide), hence its coeffect is $\{\mathsf{res}\}$. In rule (T-CONST), no variable is used.

In rule (T-FIELD-ACCESS), the coeffects are those of the receiver expression. In rule (T-FIELD-ASSIGN), the coffects of the two arguments are summed. In particular, the result of the receiver expression, of the right-side expression, and the final result, will be in sharing. For instance, we derive $\mathsf{x} :_{\{\mathsf{res}\}} \mathsf{C}, \mathsf{y} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{x.f1 = y} : \mathsf{B}$. In rule (T-NEW), analogously, the coeffects of the arguments of the constructor are summed. In particular, the results of the argument expressions and the final result will be in sharing. For instance, we derive $\mathsf{z1} :_{\{\mathsf{res}\}} \mathsf{B}, \mathsf{z2} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{new\ C(z1, z2)} : \mathsf{C}$.

In rule (T-INVK), the coeffects of the arguments are summed, after multiplying each of them with the coeffect of the corresponding parameter, where, to avoid clashes, we assume that links different from res are freshly renamed, as indicated by the notation $\equiv^{\mathsf{fr}}$. Moreover, a fresh link ℓ_i is added[8],

[7] Alternatively, variables of primitive types could be in a separate context, with no sharing coeffects.

[8] Analogously to the rule (T-APP) in Fig. 3 in the call-by-value case.

since otherwise, if the parameter is not used in the body (hence has empty coeffect), the links of the argument would be lost in the final context, see the example for rule (T-BLOCK) below.

The auxiliary function mtype now returns an enriched method type, where the parameter types are decorated with their coeffects, including the implicit parameter `this`. The condition that method bodies should be well-typed with respect to method types is extended by requiring that coeffects computed by typechecking the method body express no more sharing than those in the method type, formally: if $\mathsf{mbody}(C, m)$ and $\mathsf{mtype}(C, m)$ are defined, then $\mathsf{mbody}(C, m) = (x_1 \dots x_n, e)$, $\mathsf{mtype}(C, m) = X_0, T_1^{X_1} \dots T_n^{X_n} \to T$, and

$$\texttt{this} :_{X_0'} \mathsf{C}, x_1 :_{X_1'} T_1, \dots, x_n :_{X_n'} T_n \vdash e : T$$

$$X_i' = X_j' \neq \emptyset \text{ implies } X_i = X_j \neq \emptyset$$

holds. As an example, consider the following method:

```
class B {int f;}
class C {B f1; B f2;
  C m(B y, B z1, B z2) {this.f1=y; new C(z1,z2)}
}
```

where $\mathsf{mtype}(\mathtt{C},\mathtt{m}) = \{\ell\}, \mathsf{B}^{\{\ell\}}, \mathsf{B}^{\{\mathsf{res}\}}, \mathsf{B}^{\{\mathsf{res}\}} \to \mathsf{C}$, with $\ell \neq \mathsf{res}$. The method body is well-typed, since we derive $\texttt{this} :_{\{\ell\}} \mathsf{C}, \mathsf{y} :_{\{\ell\}} \mathsf{B}, \mathsf{z1} :_{\{\mathsf{res}\}} \mathsf{B}, \mathsf{z2} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathtt{x.f1=y; new\ C(z1,z2)} : \mathsf{C}$, with $\ell \neq \mathsf{res}$.

Consider now the method call `x.m(z,y1,y2)`. We get the following derivation:

$$\text{(T-INVK)}\ \frac{\text{(T-VAR)}\ \dfrac{}{\mathsf{x} :_{\{\mathsf{res}\}} \mathsf{C} \vdash \mathsf{x} : \mathsf{C}}\quad \text{(T-VAR)}\ \dfrac{}{\mathsf{z} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{z} : \mathsf{B}}\quad \text{(T-VAR)}\ \dfrac{}{\mathsf{y_1} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{y_1} : \mathsf{B}}\quad \text{(T-VAR)}\ \dfrac{}{\mathsf{y_2} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{y_2} : \mathsf{B}}}{\mathsf{x} :_X \mathsf{C}, \mathsf{z} :_X \mathsf{B}, \mathsf{y_1} :_Y \mathsf{B}, \mathsf{y_2} :_Y \mathsf{B} \vdash \mathtt{x.m(z,y_1,y_2)} : \mathsf{C}}$$

where $X = \{\ell', \ell_0, \ell_1\}$ and $Y = \{\mathsf{res}, \ell_2, \ell_3\}$.

The context of the call is obtained as follows

$$\begin{aligned} & \{\ell', \ell_0\} \times (\mathsf{x} :_{\{\mathsf{res}\}} \mathsf{C}) + \{\ell', \ell_1\} \times (\mathsf{z} :_{\{\mathsf{res}\}} \mathsf{B}) + \{\mathsf{res}, \ell_2\} \times (\mathsf{y_1} :_{\{\mathsf{res}\}} \mathsf{B}) + \{\mathsf{res}, \ell_3\} \times (\mathsf{y_2} :_{\{\mathsf{res}\}} \mathsf{B}) \\ = \ & (\mathsf{x} :_{\{\ell', \ell_0\}} \mathsf{C}) + (\mathsf{z} :_{\{\ell', \ell_1\}} \mathsf{B}) + (\mathsf{y_1} :_{\{\mathsf{res}, \ell_2\}} \mathsf{B}) + (\mathsf{y_2} :_{\{\mathsf{res}, \ell_3\}} \mathsf{B}) \\ = \ & \mathsf{x} :_X \mathsf{C}, \mathsf{z} :_X \mathsf{B}, \mathsf{y_1} :_Y \mathsf{B}, \mathsf{y_2} :_Y \mathsf{B} \end{aligned}$$

where ℓ' is a fresh renaming of the (method) link ℓ, and ℓ_i, $0 \leq i \leq 3$, are fresh links.

For a call `x.m(z,z,y)`, instead, we get the following derivation:

$$\text{(T-INVK)}\ \frac{\text{(T-VAR)}\ \dfrac{}{\mathsf{x} :_{\{\mathsf{res}\}} \mathsf{C} \vdash \mathsf{x} : \mathsf{C}}\quad \text{(T-VAR)}\ \dfrac{}{\mathsf{z} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{z} : \mathsf{B}}\quad \text{(T-VAR)}\ \dfrac{}{\mathsf{z} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{z} : \mathsf{B}}\quad \text{(T-VAR)}\ \dfrac{}{\mathsf{y} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{y} : \mathsf{B}}}{\mathsf{x} :_X \mathsf{C}, \mathsf{z} :_X \mathsf{B}, \mathsf{y} :_X \mathsf{B} \vdash \mathtt{x.m(z,z,y)} : \mathsf{C}}$$

where $X = \{\ell', \ell_0, \ell_1, \ell_2, \ell_3, \mathsf{res}\}$. That is, x, y, z, and the result, are in sharing (note the role of the transitive closure here).

In the examples that follow we will omit the fresh links unless necessary.

In rule (T-BLOCK), the coeffects of the expression in the declaration are multiplied by the join (that is, the union) of those of the local variable in the body and the singleton of a fresh link, and then summed with those of the body. The union with the fresh singleton is needed when the variable is not used in the body (hence has empty coeffect), since otherwise its links, that is, the information about its sharing in e, would be lost in the final context. For instance, consider the body of method `m` above, which is an abbrevation for `B unused = (this.f1=y); new (z1, z2)`. Without the join with the fresh singleton, we could derive the judgment $\texttt{this} :_{\emptyset} \mathsf{C}, \mathsf{y} :_{\emptyset} \mathsf{B}, \mathsf{z1} :_{\{\mathsf{res}\}} \mathsf{B}, \mathsf{z2} :_{\{\mathsf{res}\}} \mathsf{B} \vdash$ `B unused = (this.f1=y); new (z1, z2)` $: \mathsf{C}$, where the information that after the execution of the field assignment `this` and `y` are in sharing is lost.

Rule (T-PRIM) allows the coeffects of an expression of primitive type to be changed by removing the links with the result, as formally modeled by the product of the context with a fresh singleton coeffect. For instance, the following derivable judgment

$$\text{(T-BLOCK)}\ \frac{\text{(T-NEW)}\ \dfrac{\text{(T-CONST)}\ \dfrac{}{\emptyset \vdash 2 : \mathsf{int}}}{\emptyset \vdash \mathsf{new\ B(2)} : \mathsf{B}} \qquad \text{(T-BLOCK)}\ \dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Gamma \vdash \{\mathsf{B\,w = (x.f1 = y);\ new\ C(z,z)}\} : \mathsf{C}}}{\mathsf{x} :_{\{\ell\}} \mathsf{C}, \mathsf{y} :_{\{\ell\}} \mathsf{B} \vdash \{\mathsf{B\,z = new\ B(2);\ x.f1 = y;\ new\ C(z,z)}\} : \mathsf{C}}$$

ℓ, ℓ' fresh
$\mathsf{x} :_{\{\ell\}} \mathsf{C}, \mathsf{y} :_{\{\ell\}} \mathsf{B} = (\{\mathsf{res}\} + \{\ell'\}) \times \emptyset + \mathsf{x} :_{\{\ell\}} \mathsf{C}, \mathsf{y} :_{\{\ell\}} \mathsf{B}$
$\Gamma = (\emptyset + \{\ell\}) \times (\mathsf{x} :_{\{\mathsf{res}\}} \mathsf{C}, \mathsf{y} :_{\{\mathsf{res}\}} \mathsf{B}) + \mathsf{z} :_{\{\mathsf{res}\}} \mathsf{B} = \mathsf{x} :_{\{\ell\}} \mathsf{C}, \mathsf{y} :_{\{\ell\}} \mathsf{B}, \mathsf{z} :_{\{\mathsf{res}\}} \mathsf{B}$

$$\mathcal{D}_1 = \text{(T-FIELD-ASSIGN)}\ \frac{\text{(T-VAR)}\ \dfrac{}{\mathsf{x} :_{\{\mathsf{res}\}} \mathsf{C} \vdash \mathsf{x} : \mathsf{C}} \qquad \text{(T-VAR)}\ \dfrac{}{\mathsf{y} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{y} : \mathsf{B}}}{\mathsf{x} :_{\{\mathsf{res}\}} \mathsf{C}, \mathsf{y} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{x.f1 = y} : \mathsf{B}}$$

$$\mathcal{D}_2 = \text{(T-NEW)}\ \frac{\text{(T-VAR)}\ \dfrac{}{\mathsf{w} :_{\emptyset} \mathsf{B}, \mathsf{z} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{z} : \mathsf{B}} \qquad \text{(T-VAR)}\ \dfrac{}{\mathsf{w} :_{\emptyset} \mathsf{B}, \mathsf{z} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{z} : \mathsf{B}}}{\mathsf{w} :_{\emptyset} \mathsf{B}, \mathsf{z} :_{\{\mathsf{res}\}} \mathsf{B} \vdash \mathsf{new\ C(z,z)} : \mathsf{C}}$$

Fig. 5. Example of derivation

$\mathsf{z1} :_{\{\ell\}} \mathsf{B}, \mathsf{z2} :_{\{\ell\}} \mathsf{B} \vdash \mathsf{new\ C(z1,z2).f1.f} : \mathsf{int}$, with $\ell \neq \mathsf{res}$
shows that there is no longer a link between the result and z1, z2.

In rule (T-CONF), the coeffects of the expression and those of the memory are summed. In rule (T-MEM), a memory is well-formed in a context which is the sum of two parts. The former assigns a type to all and only references in memory, as in the standard rule in Fig. 1, and a fresh singleton coeffect. The latter sums the coeffects of the objects in memory, after multiplying each of them with that of the corresponding reference. For instance, for $\mathsf{x} \mapsto [\mathsf{y}]^{\mathsf{A}}, \mathsf{y} \mapsto [0]^{\mathsf{B}}, \mathsf{z} \mapsto [\mathsf{y}]^{\mathsf{A}}$, the former context is $\mathsf{x} :_{\{\ell_x\}} \mathsf{A}, \mathsf{y} :_{\{\ell_y\}} \mathsf{B}, \mathsf{z} :_{\{\ell_z\}} \mathsf{A}$, the latter is the sum of the three contexts $\mathsf{y} :_{\{\ell_x\}} \mathsf{A}$, $\emptyset$, and $\mathsf{y} :_{\{\ell_z\}} \mathsf{A}$. Altogether, we get $\mathsf{x} :_{\{\ell_x,\ell_y,\ell_z\}} \mathsf{A}, \mathsf{y} :_{\{\ell_x,\ell_y,\ell_z\}} \mathsf{B}, \mathsf{z} :_{\{\ell_x,\ell_y,\ell_z\}} \mathsf{A}$, expressing that the three references are connected. Note that no res link occurs in memory; indeed, there is no final result.

As an example of a more involved derivation, consider the judgment

$$\mathsf{x} :_{\{\ell\}} \mathsf{C}, \mathsf{y} :_{\{\ell\}} \mathsf{B} \vdash \{\mathsf{B\,z = new\ B(2);\ x.f1 = y;\ new\ C(z,z)}\} : \mathsf{C} \text{ where } \ell \neq \mathsf{res}.$$

Here x.f1=y; new C(z,z) is shorthand for {B w = (x.f1=y); new C(z,z)}. The derivation is in Fig. 5, where the subderivations $\mathcal{D}_1$ and $\mathcal{D}_2$ are given below for space reasons.

The rules in Fig. 4 immediately lead to an algorithm which inductively computes the coeffects of an expression. Indeed, all the rules except (T-PRIM) are syntax-directed, that is, the coeffects of the expression in the consequence are computed as a linear combination of those of the subexpressions, where the basis is the rule for variables. Rule (T-PRIM) is assumed to be *always* used in the algorithm, just once, for expressions of primitive types.

We assume there are coeffect annotations in method parameters to handle (mutual) recursion; for non-recursive methods, such coeffects can be computed (that is, in the coherency condition above, the X_is in mtype are exactly the X'_is). We leave to future work the investigation of a global fixed-point inference to compute coeffects across mutually recursive methods.

Considering again Example 2.5:

```
class B {int f; B clone [{ℓ}]() {new B(this.f)} // ℓ ≠ res
class A { B f;
  A mix [{res}](A {res} a) {this.f=a.f; a} // this, a and result linked
  A clone [{ℓ}] () {new A(this.f.clone()) } // ℓ ≠ res
}
```

```
A a1 = new A(new B(0));
A mycaps = {A a2 = new A(new B(1));
  a1.mix(a2).clone()
  // a1.mix(a2).clone().mix(a2)
}
```

The parts emphasized in gray are the coeffects which can be computed for the parameters by typechecking the body (the coeffect for `this` is in square brackets). In a real language, such coeffects would be declared by some concrete syntax, as part of the type information available to clients. From such coeffects, a client knows that the result of `mix` will be connected to both the receiver and the argument, whereas the result of `clone` will be a reference to a *fresh* portion of memory, not connected to the receiver.

By sharing coeffects, we can discriminate `a2.mix(a1).clone()` and `a1.mix(a2).clone().mix(a2)`, as desired. Indeed, for the first `mix` call, the judgment $\texttt{a1} :_{\{\text{res}\}} \texttt{A}, \texttt{a2} :_{\{\text{res}\}} \texttt{A} \vdash \texttt{a1.mix(a2)} : \texttt{A}$ holds. Then, the expression `a1.mix(a2).clone()` returns a fresh result, hence $\texttt{a1} :_{\{\ell\}} \texttt{A}, \texttt{a2} :_{\{\ell\}} \texttt{A} \vdash \texttt{a1.mix(a2).clone()} : \texttt{A}$ holds, with $\ell \neq \text{res}$. After the final call to `mix`, since `a1` and `a2` have a link in common, the operation + adds to the coeffect of `a1` the links of `a2`, including res, hence we get:

$$\texttt{a1} :_{\{\ell,\text{res}\}} \texttt{A} \vdash \texttt{\{A a2 = new A(new B(1));a1.mix(a2).clone().mix(a2)\}} : \texttt{A}$$

expressing that `a1` is linked to the result.

We now state the properties of the coeffect system for sharing.

Given $\Gamma = x_1 :_{X_1} T_1, \ldots, x_n :_{X_n} T_n$, set $\text{coeff}(\Gamma, x_i) = X_i$ and $\text{links}(\Gamma) = \bigcup_{i \in 1..n} X_i \cup \{\text{res}\}$. Finally, the *restriction* of a context $\Gamma = x_1 :_{X_1} T_1, \ldots, x_n :_{X_n} T_n$ to the set of variables $V = \{x_1, \ldots, x_m\}$, with $m \leq n$, and the set of links X, denoted $\Gamma \restriction (V, X)$, is the context $x_1 :_{Y_1} T_1, \ldots, x_m :_{Y_m} T_m$ where, for each $i \in 1..m$, $Y_i = X_i \cap X$. In the following, $\Gamma \restriction \Delta$ abbreviates $\Gamma \restriction (\text{dom}(\Delta), \text{links}(\Delta))$.

Recall that $\bowtie_\mu$ denotes the sharing relation in memory μ (Def. 2.1). The following result shows that the typing of the memory precisely captures the sharing relation.

Lemma 4.2. *If* $\Gamma \vdash \mu$, *then* $x \bowtie_\mu y$ *if and only if* $\text{coeff}(\Gamma, x) = \text{coeff}(\Gamma, y)$.

Subject reduction states that not only type but also sharing is preserved. More precisely, a reduction step may introduce new variables and new links, but the sharing between previous variables must be preserved, as expressed by the following theorem.

Theorem 4.3 (Subject reduction). *If* $\Gamma \vdash e|\mu : T$ *and* $(e, \mu) \to (e', \mu')$, *then* $\Delta \vdash e'|\mu' : T$, *for some* Δ *such that* $(\Gamma + \Delta) \restriction \Gamma = \Gamma$.

Corollary 4.4. *If* $\Gamma \vdash e|\mu : T$ *and* $(e, \mu) \to^\star (e', \mu')$, *then* $\Delta \vdash e'|\mu' : T$ *for some* Δ *such that* $(\Gamma + \Delta) \restriction \Gamma = \Gamma$.

Indeed, coeffects in $\Gamma + \Delta$ model the combined sharing before and after the computation step, hence the requirement $(\Gamma + \Delta) \restriction \Gamma = \Gamma$ ensures that, on variables in Γ, the sharing remains the same. That is, the context Δ cannot connect variables that were disconnected in Γ.

Thanks to the fact that reduction preserves (initial) sharing, we can *statically detect* lent references (Def. 2.3) and capsule expressions (Def. 2.4) just looking at coeffects, as stated below.

Theorem 4.5 (Lent reference). *If* $\Gamma \vdash e|\mu : C$, $x \in \text{dom}(\Gamma)$ *with* $\text{res} \notin \text{coeff}(\Gamma, x)$, *and* $e|\mu \to^\star y|\mu'$, *then* $x \bowtie_{\mu'} y$ *does not hold.*

Proof. By Theorem 4.3 we have $\Delta \vdash y|\mu' : C$, for some Δ such that $(\Gamma + \Delta) \restriction \Gamma = \Gamma$. By inversion, we have $y :_{\{\text{res}\}} C \vdash y : C$ with $\Delta = \Delta' + y :_{\{\text{res}\}} C$, hence $\text{res} \in \text{coeff}(\Delta, y)$. Assume $x \bowtie_{\mu'} y$. By Lemma 4.2, we have $\text{coeff}(\Delta, x) = \text{coeff}(\Delta, y)$, thus $\text{res} \in \text{coeff}(\Delta, x)$. Since $(\Gamma + \Delta) \restriction \Gamma = \Gamma$, $\text{res} \in \text{coeff}(\Gamma + \Delta, x)$ and $x \in \text{dom}(\Gamma)$, we also have $\text{res} \in \text{coeff}(\Gamma, x)$, contradicting the hypothesis. □

We write capsule(Γ) if, for each $x \in \mathrm{dom}(\Gamma)$, $\mathrm{res} \notin \mathrm{coeff}(\Gamma, x)$, that is, x is lent. The theorem above immediately implies that an expression which is typable in such a context is a capsule.

Corollary 4.6 (Capsule expression). *If* $\Gamma \vdash e|\mu : C$, *with* capsule(Γ), *and* $e|\mu \rightarrow^\star y|\mu'$, *then, for all* $x \in \mathrm{dom}(\Gamma)$, $x \bowtie_{\mu'} y$ *does not hold.*

Proof. Let $x \in \mathrm{dom}(\Gamma)$. The hypothesis capsule(Γ) means that each variable in Γ is lent that is, $\mathrm{res} \notin \mathrm{coeff}(\Gamma, x)$. Then, by Theorem 4.5, $x \bowtie_{\mu'} y$ does not hold. □

Note that, in particular, Corollary 4.6 ensures that no free variable of e can access the reachable object graph of the final result y. Notice also that assuming capsule(Γ) is the same as assuming capsule(Δ) where $\Gamma = \Delta + \Delta'$ and Δ is the context that types the expression e, because no res link can occur in the context that types the memory.

5 CASE STUDY: TYPE MODIFIERS FOR UNIQUENESS AND IMMUTABILITY

The coeffect system in the previous section tracks sharing among variables possibly introduced by reduction. In this section, we check the effectiveness of the approach to model specific language features related to sharing and mutation, taking as challenging case study those proposed by Giannini et al. [2019a,b], whose common key ideas are the following:

- types are decorated by *modifiers* mut (default, omitted in code), read, caps, and `imm` for read-only, capsule, and immutable, respectively, allowing the programmer to specify the corresponding contraints/properties for variables/parameters and method return types
- mut (resp. read) expressions can be transparently *promoted* to caps (resp. `imm`)
- caps expressions can be assigned to either mutable or immutable references.

For instance, consider the following version of Example 2.5 decorated with modifiers:

Example 5.1.

```
class B {int f; B clone [read{ℓ}] () {new B(this.f)} // ℓ ≠ res
class A { B f;
  A mix [{res}] (A {res} a) {this.f=a.f; a} // this, a and the result linked
  A clone [read{ℓ}] () {new A(this.f.clone()) } // ℓ ≠ res
}
A a1=new A(new B(0));
read A mycaps = {A a2 = new A(new B(1));
  a1.mix(a2).clone()// (1)
  // a1.mix(a2).clone().mix(a2) // (2)
}
// mycaps.f.f= 3 // (3)
a1.f.f=3 // (4)
```

The modifier of `this` in `mix` needs to be mut, whereas in `clone` it is read to allow invocations on arguments with any modifier. The result modifier in `mix` is that of the parameter `a`, chosen to be `mut` since read would have made the result of the call less usable. The result modifier of `clone` *could* be caps, but even if it is mut, the fact that there is no connection between the result and `this` is expressed by the coeffect. The difference is that with modifier caps promotion takes place when typechecking the body of the method, whereas with modifier mut it takes place at the call site.

As expected, an expression with type tagged read cannot occur as the left-hand side of a field assignment. To have the guarantee that a portion of memory is immutable, a type system should

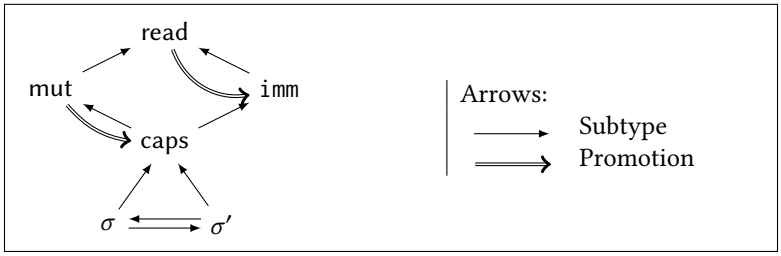

Fig. 6. Type modifiers and their relationships

be able to detect that it cannot be modified through *any* possibile reference. In the example, since `mycaps` is declared read, line (3) is ill-typed. However, if we replace line (1) with line (2), since in this case `mycaps` and `a1` share their `f` field, the same effect of line (3) can be obtained by line (4). As previously illustrated, the sharing coeffect system detects that only in the version with line (1) does `mycaps` denote a capsule. Correspondingly, in the enhanced type system in this section, `mycaps` can be correctly declared caps, hence `imm` as well, whereas this is not the case with line (2). By declaring `mycaps` of an `imm` type, the programmer has the guarantee that the portion of memory denoted by `mycaps` cannot be modified through another reference. That is, the immutability property is detected as a conjunction of the read-only restriction and the capsule property.

Assume now that `mycaps` is declared caps rather than read. Then, line (3) is well-typed. However, if `mycaps` could be assigned to both a mutable and an immutable reference, e.g:

```
A^imm imm = mycaps;
mycaps.f.f=3
```

the immutability guarantee for `imm` would be broken. For this reason, capsules can only be used linearly in the following type system.

We formalize the features illustrated above by a type-and-coeffect system built on top of that of the previous section, whose key advantage is that detection of caps and `imm` types is *straightforward* from the coeffects, through a simple *promotion*[9] rule, since they exactly express the desired properties.

Type-and-coeffect contexts are, as before, of shape $x_1 :_{X_1} T_1, \ldots, x_n :_{X_n} T_n$, where types are either primitive types or of shape C^{M}, with M modifier. We assume that fields can be declared either `imm` or mut, whereas the modifiers caps and read are only used for local variables. Besides those, which are written by the programmer in source code, modifiers include a numerable set of *seals* σ which are only internally used by the type system, as will be explained later.

Operations on coeffect contexts are lifted to type-and-coeffect contexts as in the previous case. However, there are some novelties:

- The preorder must take into account subtyping as well, defined by $T \leq T'$ if either $T = T'$ primitive type, or $T = C^{\mathrm{M}}$, $T' = C^{\mathrm{M}'}$, and $\mathrm{M} \leq \mathrm{M}'$ induced by $\sigma \leq \sigma'$, $\sigma \leq$ caps, caps $\leq$ mut, caps $\leq$ `imm`, mut $\leq$ read, `imm` $\leq$ read, see Fig. 6.
- In the sum of two contexts, denoted $\Gamma \oplus \Delta$, variables of a caps or σ type cannot occur in both; that is, they are handled *linearly*.

Combination of modifiers, denoted $\mathrm{M}[\mathrm{M}']$, is the following operation:

[9]This terminology is chosen to emphasize the analogy with promotion in linear logic.

$$
\begin{array}{lcll}
T & ::= & C^{\mathrm{M}} \mid P \mid \ldots & \text{type} \\
\mathrm{M} & ::= & \mathsf{mut} \mid \mathsf{read} \mid \mathtt{imm} \mid \mathsf{caps} \mid \sigma & \text{modifier}
\end{array}
$$

$$\text{(T-SUB)}\ \frac{\Gamma \vdash e : T' \qquad T' \leq T}{\Gamma \vdash e : T} \qquad \text{(T-VAR)}\ \frac{}{\emptyset \times \Gamma \oplus x :_{\{\mathsf{res}\}} T \vdash x : T} \qquad \text{(T-CONST)}\ \frac{}{\emptyset \triangleleft \Gamma \vdash k : P_k}$$

$$\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash e : C^{\mathrm{M}} \quad \mathsf{fields}(C) = T_1\, f_1 ; \ldots T_n\, f_n ; \quad i \in 1..n}{\Gamma \vdash e.f_i : T_i[\mathrm{M}]}$$

$$\text{(T-FIELD-ASSIGN)}\ \frac{\Gamma \vdash e : C^{\mathsf{mut}} \quad \Delta \vdash e' : T_i \quad \mathsf{fields}(C) = T_1\, f_1 ; \ldots T_n\, f_n ; \quad i \in 1..n}{\Gamma \oplus \Delta \vdash e.f_i = e' : T_i}$$

$$\text{(T-NEW)}\ \frac{\Gamma_i \vdash e_i : T_i \quad \forall i \in 1..n}{\Gamma_1 \oplus \ldots \oplus \Gamma_n \vdash \mathsf{new}\, C(e_1, \ldots, e_n) : C^{\mathsf{mut}}} \quad \mathsf{fields}(C) = T_1\, f_1 ; \ldots T_n\, f_n ;$$

$$\text{(T-INVK)}\ \frac{\Gamma_0 \vdash e_0 : C^{\mathrm{M}} \qquad \Gamma_i \vdash e_i : T_i \quad \forall i \in 1..n}{\bigoplus_{i=0}^{n} (X_i \cup \{\ell_i\}) \times \Gamma_i) \vdash e_0.m(e_1, \ldots, e_n) : T} \quad \begin{array}{l} \mathsf{mtype}(C, m) \equiv^{\mathsf{fr}} {}_{\mathrm{M}}{}^{X_0}, T_1^{X_1} \ldots T_n^{X_n} \to T \\ \ell_0, \ldots, \ell_n \text{ fresh} \end{array}$$

$$\text{(T-BLOCK)}\ \frac{\Gamma \vdash e : T \qquad \Gamma', x :_X T \vdash e' : T'}{(X \cup \{\ell\}) \times \Gamma \oplus \Gamma' \vdash \{T\, x = e;\, e'\} : T'} \quad \ell \text{ fresh}$$

$$\text{(T-IMM)}\ \frac{\Gamma \vdash e : T \quad \ell \text{ fresh}}{\{\ell\} \times \Gamma \vdash e : T} \quad T = P \text{ or } T = C^{\mathtt{imm}} \qquad \text{(T-PROM)}\ \frac{\Gamma \vdash e : C^{\mathrm{M}}}{\Gamma[\sigma] \vdash e : C^{\mathrm{M}[\mathsf{caps}]}} \quad \begin{array}{l} \mathsf{mut} \leq \mathrm{M} \\ \sigma \text{ fresh} \end{array}$$

$$\text{(T-CONF)}\ \frac{\Delta \vdash e : T \qquad \Gamma \vdash \mu}{\Delta + \Gamma \vdash e|\mu : T} \quad \mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\Gamma)$$

$$\text{(T-REF)}\ \frac{}{x :_{\{\mathsf{res}\}} C^{\mathrm{M}} \Vdash x : C^{\mathrm{M}}} \quad \mathrm{M} = \mathsf{mut} \text{ or } \mathrm{M} = \sigma \qquad \text{(T-IMM-REF)}\ \frac{}{x :_{\{\ell\}} C^{\mathtt{imm}} \Vdash x : C^{\mathtt{imm}}} \quad \ell \text{ fresh}$$

$$\text{(T-MEM-CONST)}\ \frac{}{\emptyset \Vdash k : P_k} \qquad \text{(T-OBJ)}\ \frac{\Gamma_i \Vdash v_i : T_i[\mathrm{M}] \quad \forall i \in 1..n}{\Gamma_1 + \cdots + \Gamma_n \Vdash [v_1, \ldots, v_n]^C : C^{\mathrm{M}}} \quad \mathsf{fields}(C) = T_1\, f_1 ; \ldots T_n\, f_n ;$$

$$\text{(T-MEM)}\ \frac{\Gamma_i \Vdash \mu(x_i) : C_i^{\mathrm{M}_i} \quad \forall i \in 1..n}{\Gamma_\mu + \Gamma \vdash \mu} \quad \begin{array}{l} \Gamma_\mu = x_1 :_{\{\ell_1\}} C_1^{\mathrm{M}_1}, \ldots, x_n :_{\{\ell_n\}} C_n^{\mathrm{M}_n} \\ \mathsf{dom}(\Gamma_\mu) = \mathsf{dom}(\mu) \\ \Gamma = (\{\ell_1\} \times \Gamma_1) + \ldots + (\{\ell_n\} \times \Gamma_n) \\ \ell_1, \ldots, \ell_n \text{ fresh} \end{array}$$

Fig. 7. Adding modifiers and immutability

$$\mathrm{M}[\mathrm{M}'] = \mathrm{M} \text{ if } \mathrm{M} \leq \mathtt{imm} \qquad \mathsf{mut}[\mathrm{M}] = \mathrm{M} \qquad \mathsf{read}[\mathrm{M}] = \begin{cases} \mathtt{imm} & \text{if } \mathrm{M} = \mathtt{imm} \text{ or } \mathrm{M} = \mathsf{caps} \\ \text{undefined} & \text{if } \mathrm{M} = \sigma \\ \mathsf{read} & \text{if } \mathsf{mut} \leq \mathrm{M} \end{cases}$$

Combination of modifiers is used in (T-FIELD-ACCESS) to propagate the modifier of the receiver, and in (T-PROM) to promote the type and *seal* mutable variables connected to the result, see below.

The typing rules are given in Fig. 7. We only comment on the novelties with respect to Sect. 4.

Rule (T-SUB) uses the subtyping relation defined above. For instance, an expression of type C^{caps} has the types C^{mut} and $C^{\mathtt{imm}}$ as well. In rule (T-FIELD-ACCESS), the notation $T[\mathrm{M}]$ denotes $C^{\mathrm{M}'[\mathrm{M}]}$ if $T = C^{\mathrm{M}'}$, and T otherwise, that is, if T is a primitive type. For instance, mutable fields referred to through an `imm` reference are `imm` as well. In other words, modifiers are *deep*.

In rule (T-FIELD-ASSIGN), only a mut expression can occur as the left-hand side of a field assignment. In rule (T-NEW), a constructor invocation is mut, hence mut is the default modifier of expressions of reference types. Note that the read modifier can only be introduced by variable/method declaration. The caps and `imm` modifiers, on the other hand, in addition to variable/method declaration, can be introduced by the *promotion* rule (T-PROM).

As in the previous type system, the auxiliary function mtype returns an enriched method type where the parameter types are decorated with coeffects, including the implicit parameter `this`. The condition that method bodies should be well-typed with respect to method types is exactly as in the previous type system, with only the difference that types have modifiers.

Rule (T-IMM) generalizes rule (T-PRIM) of the previous type system, allowing the links with the result to be removed, to immutable types. For instance, assuming the following variant of Example 2.2 (recall that the default modifier mut can be omitted):

```
class B {int f;}
class C {imm B f1; B f2;}
```

the following derivable judgment

$$\texttt{z1} :_{\emptyset} \texttt{B}^{\texttt{imm}}, \texttt{z2} :_{\{\ell\}} \texttt{B} \vdash \texttt{new C(z1, z2).f1} : \texttt{B}^{\texttt{imm}}, \text{ with } \ell \neq \text{res}$$

shows that there is no longer a link between the result and `z1`.

The new rule (T-PROM) plays a key role, since, as already mentioned, it detects that an expression is a capsule thanks to its coeffects, and *promotes* its type accordingly. The basic idea is that a mut (resp. read) expression can be promoted to caps (resp. `imm`) provided that there are no free variables connected to the result with modifier read or mut. However, to guarantee that type preservation holds, the same promotion should be possible for runtime expressions, which may contain free variables which actually are mut references generated during reduction. To this end, the rule allows mut variables connected to the result[10]. Such variables become *sealed* as an effect of the promotion, leading to the context $\Gamma[\sigma]$, obtained from Γ by combining modifiers of variables connected to the result with σ. Formally, if $\Gamma = x_1 :_{X_1} T_1, \ldots, x_n :_{X_n} T_n$,

$$\Gamma[\sigma] = x_1 :_{X_1} T_1', \ldots, x_n :_{X_n} T_n' \quad \text{where } T_i' = T_i[\sigma] \text{ if res} \in X_i,\ T_i' = T_i \text{ otherwise}$$

The notation $T[\sigma]$ is the same used in rule (T-FIELD-ACCESS).

This highlights once again the analogy with the promotion rule for the (graded) bang modality of linear logic [Breuvart and Pagani 2015], where, in order to introduce a modality on the right-hand side of a sequent, one has to modify the left-hand side accordingly.[11] We detail in the following how sealed variables are internally used by the type system to guarantee type preservation.

Rule (T-CONF) is as in Fig. 4. Note that we use the sum of contexts + from the previous type system, since the linear treatment of caps and σ variables is only required in source code.

Rule (T-MEM) is also analogous to that in Fig. 4. However, typechecking objects is modeled by an ad-hoc judgment $\Vdash$, where references can only be mut, `imm`, or σ (read and caps are source-only notions), and subsumption is not included. As a consequence, rule (T-OBJ) imposes that a reference reachable from an `imm` reference or field should be tagged `imm` as well, and analogously for seals.

As in the previous type system, the rules in Fig. 7 lead to an algorithm which inductively computes the coeffects of an expression. The only relevant novelty is rule (T-PROM), assumed to be applied *only when needed*, that is, when we typecheck the initialization expression of a local variable declared caps, or the argument of a method call where the corresponding parameter is declared caps. Rule (T-IMM) is applied, as (T-PRIM) before, only once, whenever an expression has either a primitive or an immutable type. Subsumption rule (T-SUB) only handles types, and can be replaced by a more

[10]Whereas read variables are still not allowed, as expressed by the fact that $\text{read}[\sigma]$ is undefined.

[11]This is just an analogy, making it precise is an interesting direction for future work.

verbose version of the rules with subtyping conditions where needed. In the other cases, rules are syntax-directed, that is, the coeffects of the expression in the consequence are computed as a linear combination of those of the subexpressions, where the basis is the rule for variables.

We illustrate now the use of seals to preserve types during reduction. For instance, consider again Example 2.2:

```
class B {int f;}
class C {B f1; B f2;}
```

$$e_0 = \{\mathsf{B\ z = new\ B(2);\ x.f1 = y;\ new\ C(z,z)}\}$$
$$\mu_0 = \{\mathsf{x} \mapsto [\mathsf{x1},\mathsf{x1}]^{\mathsf{C}}, \mathsf{x1} \mapsto [0]^{\mathsf{B}}, \mathsf{y} \mapsto [1]^{\mathsf{B}}\}$$

Expression e_0 is a *capsule* since its free variables (external resources) x and y will not be connected to the final result. Formally, set $\Delta = \mathsf{x} :_{\{\ell\}} \mathsf{C}, \mathsf{y} :_{\{\ell\}} \mathsf{B}$, with $\ell \neq \mathsf{res}$, we can derive the judgment $\Delta \vdash e_0 : \mathsf{C}^{\mathsf{mut}}$, and then apply the promotion rule (T-PROM), as shown below.

$$\text{(T-CONF)}\ \frac{\text{(T-PROM)}\ \dfrac{\Delta \vdash e_0 : \mathsf{C}^{\mathsf{mut}}}{\Delta \vdash e_0 : \mathsf{C}^{\mathsf{caps}}} \qquad \Gamma \vdash \mu_0}{\Delta + \Gamma \vdash e_0|\mu_0 : \mathsf{C}^{\mathsf{caps}}} \qquad \Gamma = \mathsf{x} :_{\{\ell_x\}} \mathsf{C}, \mathsf{x1} :_{\{\ell_x\}} \mathsf{C}, \mathsf{y} :_{\{\ell_y\}} \mathsf{B}$$

where promotion does not affect the context Δ as there are no mutable variables connected to res.

The first steps of the reduction of $e_0|\mu_0$ are as follows:

$$e_0|\mu_0 \longrightarrow e_1|\mu_1 = \{\mathsf{B\ z = w;\ x.f1 = y;\ new\ C(z,z)}\}|\mu \cup \{\mathsf{w} \mapsto [2]^{\mathsf{B}}\}$$
$$\longrightarrow e_2|\mu_1 = \mathsf{x.f1 = y;\ new\ C(w,w)}|\mu \cup \{\mathsf{w} \mapsto [2]^{\mathsf{B}}\}$$

Whereas sharing preservation, in the sense of Theorem 4.3, clearly still holds, to preserve the caps type of the initial expression the (T-PROM) promotion rule should be applicable to e_1 and e_2 as well. However, in the next steps w is a free variable connected to the result; for instance for e_1, we derive:

$$\Delta, \mathsf{w} :_{\{\mathsf{res}\}} \mathsf{B} \vdash e_1 : \mathsf{C}^{\mathsf{mut}}$$

Intuitively, e_1 is still a capsule, since w is a fresh reference denoting a closed object in memory. Formally, the promotion rule can still be applied, but variable w becomes *sealed*:

$$\text{(T-CONF)}\ \frac{\text{(T-PROM)}\ \dfrac{\Delta, \mathsf{w} :_{\{\mathsf{res}\}} \mathsf{B} \vdash e_1 : \mathsf{C}^{\mathsf{mut}}}{\Delta, \mathsf{w} :_{\{\mathsf{res}\}} \mathsf{B}^{\sigma} \vdash e_1 : \mathsf{C}^{\mathsf{caps}}} \qquad \Gamma, \mathsf{w} :_{\{\ell_w\}} \mathsf{B}^{\sigma} \vdash \mu_1}{\Delta, \mathsf{w} :_{\{\mathsf{res}\}} \mathsf{B}^{\sigma} + \Gamma \vdash e_1|\mu_1 : \mathsf{C}^{\mathsf{caps}}}$$

Capsule guarantee is preserved since a sealed reference is handled linearly, and the typing rules for memory (judgment $\Vdash$) ensure that it can only be in sharing with another one with the same seal. Moreover, the relation $\sigma \leq \sigma'$ ensures type preservation in case a group of sealed references collapses during reduction in another one, as happens with a nested promotion.

Let us denote by $\mathsf{erase}(\Gamma)$ the context obtained from Γ by erasing modifiers (hence, a context of the previous type-and-coeffect system). Subject reduction includes sharing preservation, as in the previous type system; in this case modifiers are preserved as well. More precisely, they can decrease in the type of the expression, and increase in the type of references in the context. We write $\Gamma \leq \Delta$ when, for all $x \in \mathsf{dom}(\Gamma)$, we have $\mathsf{modif}(\Gamma, x) \leq \mathsf{modif}(\Delta, x)$.

Theorem 5.2 (Subject Reduction). *If* $\Gamma \vdash e|\mu : T$ *and* $(e, \mu) \longrightarrow (e', \mu')$ *then* $\Delta \vdash e'|\mu' : T$ *for some* Δ *such that*

- $(\Gamma' + \Delta') \restriction \Gamma' = \Gamma'$, *for* $\Gamma' = \mathsf{erase}(\Gamma)$ *and* $\Delta' = \mathsf{erase}(\Delta)$;
- $\Gamma \leq \Delta$.

We now focus on properties of the memory ensured by this extended type system. First of all, we prove two lemmas characterising how the typing of memory propagates type modifiers. Recall that $\triangleright_\mu$ denotes the reachability relation in memory μ (Definition 2.6).

Lemma 5.3. *If* $\Gamma \vdash \mu$ *and* $x \triangleright_\mu y$, *then*

- $\mathsf{modif}(\Gamma, x) = \mathsf{mut}$ *implies* $\mathsf{modif}(\Gamma, y) = \mathsf{mut}$ *or* $\mathsf{modif}(\Gamma, y) = \mathtt{imm}$,
- $\mathsf{modif}(\Gamma, x) = \sigma$ *implies* $\mathsf{modif}(\Gamma, y) = \sigma$ *or* $\mathsf{modif}(\Gamma, y) = \mathtt{imm}$,
- $\mathsf{modif}(\Gamma, x) = \mathtt{imm}$ *implies* $\mathsf{modif}(\Gamma, y) = \mathtt{imm}$.

Proof. By induction on the definition of $\triangleright_\mu$.

Case $y = x$ The thesis trivially holds.

Case $\mu(x) = [v_1, \ldots, v_n]^C$, $z = v_i$ **for some** $i \in 1..n$ **and** $z \triangleright_\mu y$. From $\Gamma \vdash \mu$, by inverting rule (T-MEM), we have $\Delta \Vdash [v_1, \ldots, v_n]^C : C^{\mathrm{M}}$ with $\Gamma = \Gamma' + \{\ell\} \times \Delta$ and $\mathrm{M} = \mathsf{modif}(\Gamma, x)$. Inverting rule (T-OBJ) and either (T-REF) or (T-IMM-REF), we have $z :_X T_i[\mathrm{M}] \Vdash z : T_i[\mathrm{M}]$, with $\Delta = \Delta', z :_X T_i[\mathrm{M}]$ and $\mathsf{fields}(C) = T_1\, f_1; \ldots T_n\, f_n;$. Since $z \in \mathsf{dom}(\mu)$, T_i is of shape $C'^{\mathrm{M}'}$. We split cases on M'. If M'=imm, then $T_i[\mathrm{M}]$=imm, hence $\mathsf{modif}(\Gamma, z) = \mathtt{imm}$ and, by induction hypothesis, we get $\mathsf{modif}(\Gamma, y) = \mathtt{imm}$, as needed. If M'=mut, then $T_i[\mathrm{M}]$=M, hence we get $\mathsf{modif}(\Gamma, z) = \mathrm{M}$ and, by induction hypothesis, the thesis.

□

Lemma 5.4. *If* $\Gamma \vdash \mu$, *then* $\mathsf{coeff}(\Gamma, x)$=$\mathsf{coeff}(\Gamma, y)$ *implies* $\mathsf{modif}(\Gamma, x) = \mathsf{modif}(\Gamma, y)$.

In this refined setting, the definition of the sharing relation needs to take into account modifiers. Indeed, if intuitively two references are in sharing when a mutation of either of the two affects the other, then no sharing should be propagated through immutable references. To do so, we need to assume a well-typed memory in order to know modifiers of references.[12]

Definition 5.5 (Sharing in memory with modifiers). The *sharing relation* in memory $\Gamma \vdash \mu$, denoted by $\bowtie_{\Gamma,\mu}$, is the smallest equivalence relation on $\mathsf{dom}(\mu)$ such that:

$x \bowtie_{\Gamma,\mu} y$ if $\mu(x) = [v_1, \ldots, v_n]^C$, $\mathsf{modif}(\Gamma, x), \mathsf{modif}(\Gamma, y) \leq \mathsf{mut}$ and $y = v_i$ for some $i \in 1..n$

Again, for a well-typed memory, coeffects characterize the sharing relation exactly.

Proposition 5.6. *If* $\Gamma \vdash \mu$, *then* $x \bowtie_{\Gamma,\mu} y$ *iff* $\mathsf{coeff}(\Gamma, x) = \mathsf{coeff}(\Gamma, y)$, *for all* $x, y \in \mathsf{dom}(\mu)$.

In the extended type system, we can detect capsule expressions from the modifier, without looking at coeffects of free variables, proving that the result of a caps expression is not in sharing with the initial mutable variables.

Theorem 5.7 (Capsule expression). *If* $\Gamma \vdash e|\mu : C^{\mathsf{caps}}$, *and* $e|\mu \rightarrow^\star y|\mu'$, *then there exists* Γ' *such that* $\Gamma' \vdash \mu'$, $\Gamma \leq \Gamma'$ *and, for all* $x \in \mathsf{dom}(\mu)$, $x \bowtie_{\Gamma',\mu'} y$ *implies* $\mathsf{modif}(\Gamma, x) \leq \mathsf{modif}(\Gamma', x) \neq \mathsf{mut}$.

Proof. By Theorem 5.2, we get $\Delta \vdash y|\mu' : C^{\mathsf{caps}}$ with $\Gamma \leq \Delta$. By inverting rule (T-CONF), we get $\Delta_1 \vdash y : C^{\mathsf{caps}}$ and $\Delta_2 \vdash \mu'$, with $\Delta = \Delta_1 + \Delta_2$ and $\Gamma \leq \Delta_2$, as $\mathsf{modif}(\Delta_2, z) = \mathsf{modif}(\Delta, z)$ for all $\in \mathsf{dom}(\Delta)$. Since $y \in \mathsf{dom}(\mu')$, it cannot have modifier caps, hence $\Delta_1 \vdash y : C^{\mathsf{caps}}$ holds by rule (T-PROM) or (T-SUB). This implies $\Delta_1 = \emptyset \times \Delta', y :_{\{\mathsf{res}\}} C^\sigma$ and so $\mathsf{modif}(\Delta, y) = \mathsf{modif}(\Delta_2, y) = \mathsf{modif}(\Delta_1, y) = \sigma$. Set $\Gamma' = \Delta_2$. By Proposition 5.6 and Lemma 5.4, $x \bowtie_{\Gamma',\mu'} y$ implies $\mathsf{modif}(\Gamma', x) = \sigma$, thus $\mathsf{modif}(\Gamma, x) \leq \mathsf{modif}(\Delta, x) = \mathsf{modif}(\Gamma', x) \neq \mathsf{mut}$, hence the thesis. □

It is important to notice that the notion of capsule expression in Theorem 5.7 is different from the previous one (Definition 2.4), as we now have imm references. In particular, the previous notion prevented any access to the reachable object graph of the result from free variables, since, without modifiers, any access to a portion of memory can modify it. Here, instead, this is no longer true, hence the notion of capsule allows mutable references to access the reachable object graph of the result of a capsule expression, but only through imm references. Indeed, if two references access the same non-imm reference, they are necessarily in sharing, as shown below.

[12] Actually, we do not need the full typing information, having just modifiers would be enough.

Proposition 5.8. *Let* $\Gamma \vdash \mu$. *If* $x \triangleright_\mu z$ *and* $y \triangleright_\mu z$ *and* $\mathsf{modif}(\Gamma, z) \neq \mathtt{imm}$, *then* $x \bowtie_{\Gamma,\mu} y$.

Proof. We first show that $x \bowtie_\mu z$. The proof is by induction on the definition of $\triangleright_\mu$.

Case $x = z$ The thesis trivially holds by reflexivity of $\bowtie_{\Gamma,\mu}$.

Case $\mu(x) = [v_1, \ldots, v_n]^C$, $x' = v_i$ **for some** $i \in 1..n$ **and** $x' \triangleright_\mu z$ We know that $\mathsf{modif}(\Gamma, x)$, $\mathsf{modif}(\Gamma, x') \leq \mathsf{mut}$ since, by Lemma 5.3, $\mathsf{modif}(\Gamma, x') = \mathtt{imm}$ (or $\mathsf{modif}(\Gamma, x) = \mathtt{imm}$) would imply $\mathsf{modif}(\Gamma, z) = \mathtt{imm}$ which is a contradiction. Then, by Definition 5.5, we have $x \bowtie_{\Gamma,\mu} x'$ and by induction hypothesis, we get $x' \bowtie_{\Gamma,\mu} z$; then we get $x \bowtie_{\Gamma,\mu} z$ by transitivity of $\bowtie_{\Gamma,\mu}$.

By the same argument, we also get $y \bowtie_{\Gamma,\mu} z$. Then, by transitivity of $\bowtie_{\Gamma,\mu}$, we get the thesis. □

Corollary 5.9. *If* $\Gamma \vdash e|\mu : C^{\mathsf{caps}}$, *and* $e|\mu \rightarrow^\star y|\mu'$, *then there exists* Γ' *such that* $\Gamma' \vdash \mu'$, $\Gamma \leq \Gamma'$ *and, for all* $x \in \mathsf{dom}(\mu)$, $\mathsf{modif}(\Gamma', x) = \mathsf{mut}$ *and* $x \triangleright_{\mu'} z$ *and* $y \triangleright_{\mu'} z$ *imply* $\mathsf{modif}(\Gamma', z) = \mathtt{imm}$.

Proof. By Theorem 5.7, we get $\Gamma' \vdash \mu'$ and $x \bowtie_{\Gamma',\mu'} y$ imply $\mathsf{modif}(\Gamma', x) \neq \mathsf{mut}$. Suppose $\mathsf{modif}(\Gamma', z) \neq \mathtt{imm}$, then, by Proposition 5.8, we get $x \bowtie_{\Gamma',\mu'} y$, hence $\mathsf{modif}(\Gamma', x) \neq \mathsf{mut}$, which contradicts the hypothesis. Therefore, $\mathsf{modif}(\Gamma', z) = \mathtt{imm}$. □

In the extended type system, we can also nicely characterize the property guaranteed by the `imm` references. Notably, the reachable object graph of an `imm` modifier cannot be modified during the execution. We first show that fields of an `imm` reference cannot change in a single computation step.

Lemma 5.10. *If* $\Gamma \vdash e|\mu : T$, *and* $\mathsf{modif}(\Gamma, x) = \mathtt{imm}$, *and* $e|\mu \rightarrow e'|\mu'$, *then* $\mu(x) = \mu'(x)$.

Proof. By induction on reduction rules. The key case is rule (field-assign). We have $e = y.f = v$ and $\Gamma \vdash y.f = v|\mu : T$. Let $\mathsf{modif}(\Gamma, y) = \mathsf{M}$. Either rule (T-field-assign) was the last rule applied, or one of the non syntax-directed rules was applied after (T-field-assign). In the former case $\mathsf{M} = \mathsf{mut}$ or $\mathsf{M} = \mathsf{caps}$ if rule (T-Sub) was applied before (T-field-assign). In the latter case M could only be equal to the previous modifier or $\mathsf{M} = \sigma$ if rule (T-Prom) was applied and the previous modifier was mut. Therefore, $y \neq x$ and so we have the thesis. For all other computational rules the thesis is immediate as they do not change the memory, and for (ctx) the thesis immediately follows by induction hypothesis. □

Thanks to Lemma 5.3, we can show that the reachable object graph of an `imm` reference contains only `imm` references. Hence, by the above lemma we can characterise `imm` references as follows:

Theorem 5.11 (Immutable reference). *If* $\Gamma \vdash e|\mu : T$, $\mathsf{modif}(\Gamma, x) = \mathtt{imm}$, *and* $e|\mu \rightarrow^\star e'|\mu'$, *then* $x \triangleright_\mu y$ *implies* $\mu(y) = \mu'(y)$.

Proof. By induction on the definition of $\rightarrow^\star$

Case $e|\mu = e'|\mu'$ The thesis trivially holds.

Case $e|\mu \rightarrow e_1|\mu_1 \rightarrow^\star e'|\mu'$ Since $\mathsf{modif}(\Gamma, x) = \mathtt{imm}$, for all y such that $x \triangleright_\mu y$, by Lemma 5.3 $\mathsf{modif}(\Gamma, y) = \mathtt{imm}$, hence, by Lemma 5.10, $\mu_1(y) = \mu(y)$. Therefore, it is easy to check that $x \triangleright_\mu y$ implies $x \triangleright_{\mu_1} y$. By Theorem 5.2, $\Delta \vdash e_1|\mu_1 : T$ and $\mathsf{modif}(\Gamma, x) \leq \mathsf{modif}(\Delta, x)$, hence $\mathsf{modif}(\Delta, x) = \mathtt{imm}$. Then, by induction hypothesis, $\mu'(y) = \mu_1(y)$, hence the thesis. □

6 EXPRESSIVE POWER

We discuss the expressive power of the type-and-coeffect system in Sect. 5, comparing it with the two most closely related proposals by Gordon et al. [2012] and by Clebsch [2017]; Clebsch et al. [2015], abbreviated as Gordon et al. and Pony, respectively. The takeaway is that our promotion mechanism is much more powerful than their recovery, since sharing is taken into account; on the other hand, the expressive power allowed by some of their annotations on fields is beyond the scope of this paper. We assume a syntax enriched by the usual programming constructs.

Before the work in Gordon et al., the capsule property was only ensured in simple situations, such as using a primitive deep clone operator, or composing subexpressions with the same property. The type system in Gordon et al. has been an important step, being the first to introduce *recovery*. That is, this type system contains two typing rules which allow recovering isolated[13] or immutable references from arbitrary code checked in contexts containing only isolated or immutable variables. Such rules are rephrased below in our style for better comparison.

$$\text{(T-RECOV-ISO)}\ \frac{\Gamma \vdash e : C^{\mathsf{mut}}}{\Gamma \vdash e : C^{\mathsf{caps}}}\ \mathsf{IsoOrImm}(\Gamma) \qquad \text{(T-RECOV-IMM)}\ \frac{\Gamma \vdash e : C^{\mathsf{read}}}{\Gamma \vdash e : C^{\mathtt{imm}}}\ \mathsf{IsoOrImm}(\Gamma)$$

where $\mathsf{IsoOrImm}(\Gamma)$ means that, for all $x : C^{\mathrm{M}}$ in Γ, $\mathrm{M} \leq \mathtt{imm}$.

As the reader can note, this is exactly in the spirit of coeffects, since typechecking also takes into account the way the *surrounding context* is used. By these rules Gordon et al. typechecks, e.g., the following examples, assuming the language has threads with a parallel operator:

```
isolated IntList l1 = ...
isolated IntList l2 = ...
l1.map(new Incrementor()); || l2.map(new Incrementor());
```

The two threads do not interfere, since they operate and can mutate disjoint object graphs.

```
isolated IntBox increment(isolated IntBox b){
  b.value++;//b converted to mut by subtyping
  return b//convert b *back* to isolated by recovery
}
```

An isolated object can be mutated[14], and then isolation can be recovered, since the context only contains isolated or immutable references.

In Pony, the ideas of Gordon et al. are extended to a richer set of modifiers. In their terminology `val` is immutable, `ref` is mutable, `box` is read-only. An ephemeral isolated reference `iso^` is similar to a caps reference in our calculus, whereas non ephemeral `iso` references are more similar to the isolated fields discussed below. Finally, `tag` only allows object identity checks and asynchronous method invocation, and `trn` (transition) is a subtype of `box` that can be converted to `val`, providing a way to create values without using isolated references. The last two modifiers have no equivalent in Gordon et al. or our work.

The type-and-coeffect-system in Sect. 5 shares with Gordon et al. and Pony the modifiers mut, `imm`, read, and caps with their subtyping relation, a similar operation to combine modifiers, and the key role of recovery. However, rule (T-PROM) is much more powerful than the recovery rules reported above, which definitely forbid read and mut variables in the context. Rule (T-PROM), instead, allows such variables when they are not connected to the final result, as smoothly derived from coeffects which compute sharing. For instance, with the classes of Example 2.2, the following two examples would be ill-typed in Gordon et al. and Pony:

```
caps C es1 = {B z = new B(2); x.f1=y; new C(z,z)}
caps C es2 = {B z = new B(y.f=y.f+1); new C(z,z) }
```

Below is the corresponding Pony code.

```
class B
  var f: U64
  new create(ff:U64) => f=ff
class C
```

[13]Their terminology for capsule.
[14]We say that the capsule is *opened*, see in the following.

```
  var f1: B
  var f2: B
  new create(ff1: B ref, ff2: B ref) =>  f1=ff1; f2=ff2
var x: B ref = ...
var y: B ref = ...
var es1: C iso = recover iso var z = B(2); x.f1=y; C(z,z) end//ERROR
var es2: C iso = recover iso var z = B(y.f=y.f+1); C(z,z) end//ERROR
```

For comparison on a more involved example, let us add to class A of Example 5.1 the method nonMix that follows:

```
  A nonMix [{ℓ}] (A {res} a) {this.f.f=a.f.f; a} // ℓ ≠ res
```

Consider the following code:

```
A a1= new A(new B(0));
caps A mycaps = {A a2 = new A(new B(1));
  a1.mix(a2).clone() // (1)
  // a1.mix(a2).clone().mix(a2) // (2)
  // a1.nonMix(a2) // (3)
}
```

The corresponding Pony code is as follows:

```
class B
  var f:U64
  new create(ff:U64) => f=ff
  fun box clone():B iso^ => recover B(f) end
class A
  var f:B
  new create(ff:B) => f=ff
  fun ref mix(a:A):A => this.f=a.f; a
  fun ref nonMix(a:A):A => f.f=a.f.f; a
  fun box clone():A iso^ =>var x:B iso=f.clone();recover A(consume x) end
var a1 = A(B(0))
var a2 = A(B(1)); var l1:A iso = a1.mix(a2).clone() // (1)
var l2:A iso=recover var a2=A(B(1));a1.mix(a2).clone().mix(a2) end//(2)
var l3:A iso= recover var a2 = A(B(1)); a1.nonMix(a2) end // (3)
```

As in our approach, Pony is able to discriminate line (1) from line (2), causing code to be well-typed and ill-typed, respectively. However, to do so, Pony needs an explicit modifier `iso^` in the return type of `clone`, whereas, as noted after the code of Example 5.1, in our approach the return type of `clone` can be mut, since the fact that there is no connection between the result and `this` is expressed by the coeffect. Moreover, in order to be able to obtain an `iso` from the `clone` method, Pony needs to insert explicit `recover` instructions. In the case of class `A` where the field is an object, Pony needs to explicitly use `consume` to ensure uniqueness, whereas in our approach promotion takes place implicitly and uniqueness is ensured by linearity. Finally, Pony rejects line (3) as well, whereas, in our approach, this expression is correctly recognized to be a capsule, since the external variable `a1` is modified, but not connected to the final result.

Moreover, our type system can prevent sharing of parameters, something which is not possibile in Gordon et al. and Pony. The following method takes two teams, `t1` and `t2`, as parameters. Both want to add a reserve player from their respective lists `p1` and `p2`, sorted with best players first. To keep the game fair, the two reserve players can only be added if they have the same skill level.

```
static void addPlayer(Team {ℓ} t1, Team {ℓ′} t2, Players {ℓ} p1, Players {ℓ′} p2)
{/*ℓ ≠ ℓ′*/} {while(true){
 if(p1.isEmpty()||p2.isEmpty()) {/*error*/}
 if(p1.top().skill==p2.top().skill){t1.add(p1.top());t2.add(p2.top());}
 else{removeMoreSkilled(p1,p2);}
}
```

The sharing coeffects express that each team can only add players from its list of reserve players.

As mentioned at the beginning of the section, an important feature supported by Gordon et al. and Pony, and not by our type system, are isolated fields. To ensure that accessing an isolated field will not introduce aliasing, they use an ad-hoc semantics, called *destructive read*, see also Boyland [2010]. In Gordon et al., an isolated field can only be read by a command `x=consume(y.f)`, assigning the value to `x` and updating the field to `null`. Pony supports the command `(consume x)`, with the semantics that the reference becomes empty. Since fields cannot be empty, they cannot be arguments of `consume`. By relying on the fact that assignment returns the left-hand side value, in Pony one writes `x=y.f=(consume z)`, with `z` isolated. In this way, the field value is assigned to `x`, and the field is updated to a new isolated reference.

We prefer to avoid destructive reads since they can cause subtle bugs, see Giannini et al. [2019b] for a discussion. We leave to future work the development of an alternative solution, notably investigating how to extend our affine handling of caps variables to fields. Concerning this point, another feature allowing more flexibility in Gordon et al. and Pony is that iso variables can be "consumed" only once, but accessed more than once. For example in Pony we can write `var c: C iso=recover var z=B(2); C(z,z) end; c.f1=recover B(1) end;c.f2=recover B(1) end`. To achieve this in our type system, one needs to explicitly open the capsule by assigning it to a local mutable variable, modify it and finally apply the promotion to recover the capsule property.

7 RELATED WORK

7.1 Coeffect Systems

Coeffects were first introduced by Petricek et al. [2013] and further analyzed by Petricek et al. [2014]. In particular, Petricek et al. [2014] develops a generic coeffect system which augments the simply-typed λ-calculus with context annotations indexed by *coeffect shapes*. The proposed framework is very abstract, and the authors focus only on two opposite instances: structural (per-variable) and flat (whole context) coeffects, identified by specific choices of context shapes.

Most of the subsequent literature on coeffects focuses on structural ones, for which there is a clear algebraic description in terms of semirings. This was first noticed by Brunel et al. [2014], who developed a framework for structural coeffects for a functional language. This approach is inspired by a generalization of the exponential modality of linear logic, see, e.g., Breuvart and Pagani [2015]. That is, the distinction between linear and unrestricted variables of linear systems is generalized to have variables decorated by coeffects, or *grades*, that determine how much they can be used. In this setting, many advances have been made to combine coeffects with other programming features, such as computational effects [Dal Lago and Gavazzo 2022; Gaboardi et al. 2016; Orchard et al. 2019], dependent types [Atkey 2018; Choudhury et al. 2021; McBride 2016], and polymorphism [Abel and Bernardy 2020]. A fully-fledged functional programming language, called Granule, has been presented by Orchard et al. [2019], inspired by the principles of coeffect systems.

As already mentioned, McBride [2016] and Wood and Atkey [2022] observed that contexts in a structural coeffect system form a module over the semiring of grades, event though they do not use this structure in its full generality, restricting themselves to free modules, that is, to structural coeffect systems. This algebraic structure nicely describes operations needed in typing rules, and

we believe it could be a clean framework for coeffect systems beyond structural ones. Indeed, the sharing coeffect system in this paper provides a non-structural instance.

7.2 Type Systems Controlling Sharing and Mutation

The literature on type systems controlling sharing and mutation is vast. In Sect. 6 we provided a comparison with the most closely related approaches. We briefly discuss here other works.

The approach based on modifiers is extended in other proposals [Castegren and Wrigstad 2016; Haller and Odersky 2010] to compositions of one or more *capabilities*. 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 modifiers, as accessing different parts of an object through different capabilities in the same type gives different properties. By using capabilities it is possible to obtain an expressivity similar to our type system, although with different sharing notions and syntactic constructs. For instance, the *full encapsulation* notion by Haller and Odersky [2010], apart from the fact that sharing of immutable objects is not allowed, is equivalent to our caps guarantee. Their model has a higher syntactic/logic overhead to explicitly track regions. As for all work preceding Gordon et al. [2012], objects need to be born `unique` and the type system permits manipulation of data preserving their uniqueness. With recovery/promotion, instead, we can use normal code designed to work on conventional shared data, and then recover uniqueness.

An alternative approach to modifiers to restrict the usage of references is that of *ownership*, based on *enforcing invariants* rather than deriving properties. We refer to the recent work of Milano et al. [2022] for an up-to-date survey. The Rust language, considering its "safe" features [Jung et al. 2018], belongs to this family as well, and uses ownership for memory management. In Rust, all references which support mutation are required to be affine, thus ensuring a unique entry point to a portion of mutable memory. This relies on a relationship between linearity and uniqueness recently clarified by Marshall et al. [2022], which proposes a linear calculus with modalities for non-linearity and uniqueness with a somewhat dual behaviour. In our approach, instead, the capsule concept models an efficient *ownership transfer*. In other words, when an object x is "owned" by y, it remains always true that y can only be accessed through x, whereas the capsule notion is dynamic: a capsule can be "opened", that is, assigned to a standard reference and modified, since we can always recover the capsule guarantee.[15].

We also mention that, whereas in this paper all properties are *deep*, that is, inherited by the reachable object graph, most ownership approaches allows one to distinguish subparts of the reachable object graph that are referred to but not logically owned. This viewpoint has some advantages, for example Rust uses ownership to control deallocation without a garbage collector.

8 CONCLUSION

The main achievement of this paper is to show that sharing and mutation can be tracked by a coeffect system, thus reconciling two distinct views in the literature on type systems for mutability control: substructural type systems, and graph-theoretic properties on heaps. Specifically, the contributions of the paper are the following:

- a minimal framework formalizing ingredients of coeffect systems
- a coeffect system, for an imperative Java-like calculus, where coeffects express *links* among variables and with the final result introduced by the execution
- an enhanced type system modeling complex features for uniqueness and immutability.

[15]Other work in the literature supports ownership transfer, see, e.g., Müller and Rudich [2007] and Clarke and Wrigstad [2003], however not of the whole reachable object graph.

The enhanced type system (Sect. 5) cleanly integrates and formalizes language designs by Giannini et al. [2019b] and Giannini et al. [2019a], as detailed below:

- Giannini et al. [2019b] supports promotion through a very complex type system; moreover, sharing of two variables/parameters cannot be prevented, as, e.g., in the example on page 24.
- Giannini et al. [2019a] has a more refined tracking of sharing allowing us to express this example, but does not handle immutability.
- In both works, significant properties are expressed and proved with respect to a non-standard reduction model where memory is encoded in the language itself.

Each of the contributions of the paper opens interesting research directions. The minimal framework we defined, modeling coeffect contexts as modules, includes structural coeffect systems, where the coeffect of each variable can be computed independently (that is, the module operators are defined pointwise), and coeffect systems such as those in this paper, which can be considered *quasi-structural*. Indeed, coeffects cannot be computed per-variable (notably, the sum and multiplication operator of the module are *not* defined pointwise), but can still be expressed by annotations on single variables. This also shows a difference with existing graded type systems explicitly derived from bounded linear logic, which generally consider purely structural (that is, computable per-variable) grades.[16]. In future work we plan to develop the metatheory of the framework, and to investigate its appropriateness both for other quasi-structural cases, and for coeffects which are truly *flat*, that is, cannot be expressed on a per-variable basis. This could be coupled with the design of a λ-calculus with a generic module-based coeffect system, substantially revising [Petricek et al. 2014].

In the type system in Sect. 5, coeffects and modifiers are distinct, yet interacting, features. We will investigate a framework where modifiers, or, more generally, *capabilities* [Gordon 2020; Gordon et al. 2012; Haller and Odersky 2010], are formalized as graded modal types, which are, roughly, types annotated with coeffects (grades) [Brunel et al. 2014; Dal Lago and Gavazzo 2022; Orchard et al. 2019], thus providing a formal foundation for the "capability" notion in the literature. A related interesting question is which kinds of modifier/capability can be expressed *purely* as coeffects. The read-only property, for instance, could be expressed by enriching the sharing coeffect with a Read/Write component (Read by default), so that in a field assignment, variables connected to `res` in the context of the left-hand expression are marked as Write.

Concerning the specific type system in Sect. 5, additional features are necessary to have a more realistic language. Two important examples are: suitable syntactic sugar for coeffect annotations in method types and relaxation of coeffects when redefining methods in the presence of inheritance.

Finally, an interesting objective in the context of Java-like languages is to allow variables (notably, method parameters) to be annotated by user-defined coeffects, written by the programmer by extending a predefined abstract class, in the same way user-defined exceptions extend the `Exception` predefined class. This approach would be partly similar to that of Granule [Orchard et al. 2019], where, however, coeffects cannot be extended by the programmer. A first step in this direction is presented by Bianchini et al. [2022a].

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees who provided useful and detailed comments on a previous version of the paper. We also thank Nicholas Webster for help with the Pony examples, and Peter Neuss for proofreading the paper. This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X) and has the financial support of the Università del Piemonte Orientale.

[16]This corresponds to a free module, while the module of sharing coeffects is not free (just a retraction of a free one).

REFERENCES

Andreas Abel and Jean-Philippe Bernardy. 2020. A unified view of modalities in type systems. *Proceedings of ACM on Programming Languages* 4, ICFP (2020), 90:1–90:28. https://doi.org/10.1145/3408972

Paulo Sérgio Almeida. 1997. Balloon Types: Controlling Sharing of State in Data Types. In *European Conference on Object-Oriented Programming, ECOOP 1997 (Lecture Notes in Computer Science, Vol. 1241)*. Springer, 32–59.

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *IEEE Symposium on Logic in Computer Science, LICS 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM Press, 56–65. https://doi.org/10.1145/3209108.3209189

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2022a. A Java-like calculus with user-defined coeffects. In *ICTCS'22 - Italian Conf. on Theoretical Computer Science*. To appear.

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, and Marco Servetto. 2022b. Coeffects for sharing and mutation. *CoRR* abs/2209.07439 (2022). https://doi.org/10.48550/arXiv.2209.07439

John Boyland. 2010. Semantics of Fractional Permissions with Nesting. *ACM Transactions on Programming Languages and Systems* 32, 6 (2010).

Flavien Breuvart and Michele Pagani. 2015. Modelling Coeffects in the Relational Semantics of Linear Logic. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015 (LIPIcs, Vol. 41)*, Stephan Kreutzer (Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 567–581. https://doi.org/10.4230/LIPIcs.CSL.2015.567

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *European Symposium on Programming, ESOP 2013 (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *European Conference on Object-Oriented Programming, ECOOP 2016 (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 5:1–5:26.

Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A graded dependent type system with a usage-aware semantics. *Proceedings of ACM on Programming Languages* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434331

David Clarke and Tobias Wrigstad. 2003. External Uniqueness is Unique Enough. In *European Conference on Object-Oriented Programming, ECOOP 2003 (Lecture Notes in Computer Science, Vol. 2473)*, Luca Cardelli (Ed.). Springer, 176–200.

Sylvan Clebsch. 2017. *'Pony': co-designing a type system and a runtime*. Ph. D. Dissertation. Imperial College London, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.769552

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela (Eds.). ACM Press, 1–12.

Ugo Dal Lago and Francesco Gavazzo. 2022. A relational theory of effects and coeffects. *Proceedings of ACM on Programming Languages* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498692

Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *European Conference on Object-Oriented Programming, ECOOP 2007 (Lecture Notes in Computer Science, Vol. 4609)*, Erik Ernst (Ed.). Springer, 28–53.

Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *ACM International Conference on Functional Programming, ICFP 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM Press, 476–489. https://doi.org/10.1145/2951913.2951939

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *European Symposium on Programming, ESOP 2013 (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

Paola Giannini, Tim Richter, Marco Servetto, and Elena Zucca. 2019a. Tracing sharing in an imperative pure calculus. *Science of Computer Programming* 172 (2019), 180–202. https://doi.org/10.1016/j.scico.2018.11.007 Extended version, *CoRR*, https://arxiv.org/abs/1803.05838.

Paola Giannini, Marco Servetto, Elena Zucca, and James Cone. 2019b. Flexible recovery of uniqueness and immutability. *Theoretical Computer Science* 764 (2019), 145–172. https://doi.org/10.1016/j.tcs.2018.09.001 Extended version, *CoRR*, https://arxiv.org/abs/1807.00137.

Colin S. Gordon. 2020. Designing with Static Capabilities and Effects: Use, Mention, and Invariants (Pearl). In *European Conference on Object-Oriented Programming, ECOOP 2020 (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:25. https://doi.org/10.4230/LIPIcs.ECOOP.2020.10

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

Philipp Haller and Martin Odersky. 2010. Capabilities for uniqueness and borrowing. In *European Conference on Object-Oriented Programming, ECOOP 2010 (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 354–378.

John Hogg. 1991. Islands: Aliasing Protection in Object-oriented Languages. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991*, Andreas Paepcke (Ed.). ACM Press, 271–285.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 1999. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*. ACM Press, 132–146. https://doi.org/10.1145/320384.320395

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proceedings of ACM on Programming Languages* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Daniel Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *European Symposium on Programming, ESOP 2022 (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 346–375. https://doi.org/10.1007/978-3-030-99336-8_13

Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Mae Milano, Andrew C. Meyers, and Joshua Turcotti. 2022. A Flexible Type System for Fearless Concurrency. To appear.

Peter Müller and Arsenii Rudich. 2007. Ownership transfer in universe types. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 2007*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM Press, 461–478.

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proceedings of ACM on Programming Languages* 3, ICFP (2019), 110:1–110:30. https://doi.org/10.1145/3341714

Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. 2013. Coeffects: Unified Static Analysis of Context-Dependence. In *Automata, Languages and Programming, ICALP 2013 (Lecture Notes in Computer Science, Vol. 7966)*, Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg (Eds.). Springer, 385–397. https://doi.org/10.1007/978-3-642-39212-2_35

Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. In *ACM International Conference on Functional Programming, ICFP 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM Press, 123–135. https://doi.org/10.1145/2628136.2628160

Marco Servetto, David J. Pearce, Lindsay Groves, and Alex Potanin. 2013. Balloon Types for Safe Parallelisation over Arbitrary Object Graphs. In *WODET 2014 - Workshop on Determinism and Correctness in Parallel Programming*.

James Wood and Robert Atkey. 2022. A Framework for Substructural Type Systems. In *European Symposium on Programming, ESOP 2022 (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 376–402. https://doi.org/10.1007/978-3-030-99336-8_14