Theoretical Computer Science 971 (2023) 114063

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

A Java-like calculus with heterogeneous coeffects [☆]

Riccardo Bianchini [a], Francesco Dagnino [a], Paola Giannini [b,*], Elena Zucca [a]

[a] DIBRIS, University of Genoa, Italy
[b] DiSSTE, University of Eastern Piedmont, Vercelli, Italy

ARTICLE INFO

Article history:
Received 9 January 2023
Received in revised form 29 June 2023
Accepted 4 July 2023
Available online 11 July 2023

Keywords:
Type systems
Operational semantics
Java-like languages

ABSTRACT

We propose a Java-like calculus where declared variables can be annotated by *coeffects* specifying constraints on their use, e.g., affinity or privacy levels. Such coeffects are *heterogeneous*, in the sense that different kinds of coeffects can be used in the same program; combining coeffects of different kinds leads to the trivial coeffect. We prove subject reduction, which includes preservation of coeffects, and show several examples. In a Java-like language, coeffects can be expressed in the language itself, as expressions of user-defined classes.

1. Introduction

Type-and-coeffect systems [21,8,2,12,13,19,9,10] are type systems where the typing judgment takes the form $x_1 :_{r_1} T_1, \ldots, x_n :_{r_n} T_n \vdash e : T$, with $r_1, \ldots, r_n$ *coeffects* (also called *grades*, using the alternative terminology *graded type system*) modeling how the corresponding variables are used in e. For instance, coeffects of shape $r ::= 0 \mid 1 \mid \omega$ trace when a variable is either not used, or used at most once, or used in an unrestricted way, respectively, in the expression e. In this way, functions, e.g., λx:int.5, λx:int.x, and λx:int.$x + x$, which have the same type in the simply-typed lambda calculus, can be distinguished by adding coeffect annotations: λx:int[0].5, λx:int[1].x, and λx:int[ω].$x + x$. Other typical examples are counting usages (coeffects are natural numbers), and privacy levels. Coeffects usually form a semiring, specifying *sum* $+$, *multiplication* $\cdot$, and **0** and **1** constants, satisfying some natural axioms. Some kind of order relation is generally required as well.

This approach has been exploited to a fully-fledged programming language in Granule [19], a functional language equipped with a type-and-coeffect system, hence allowing the programmer to write function declarations as those above. In Granule, different kinds of coeffects can be used at the same time, including naturals for counting usages, privacy levels, intervals, infinity, and products of coeffects; however, the available coeffects are fixed once and for all.

In this paper, we aim at providing a similar support in Java-like languages, by allowing the programmer to write coeffect annotations in variable declarations. As in Granule, *heterogeneous* coeffects can coexist in the same program. When combining coeffects of different kinds, we take the simple solution that this leads to the trivial coeffect. (We will investigate in future work how to provide a general form of combination, see the Conclusion.) This is formally modeled by a construction which, given a family of coeffect algebras, indexed over a set of *kinds*, returns a coeffect algebra where coeffects are decorated by their original kind. We prove subject reduction, which includes preservation of coeffects.

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.
* Principal corresponding author.

E-mail addresses: riccardo.bianchini@edu.unige.it (R. Bianchini), francesco.dagnino@dibris.unige.it (F. Dagnino), paola.giannini@uniupo.it (P. Giannini), elena.zucca@unige.it (E. Zucca).

https://doi.org/10.1016/j.tcs.2023.114063

In a Java-like language, coeffects desired for a specific application could be expressed in the language itself. More in detail, coeffect annotations could be expressions of *coeffect classes*, that is, classes providing methods corresponding to the ingredients of a coeffect algebra. In this way, the programmer could write user-defined coeffects desired for a specific application, rather than rely on a fixed set of coeffects as in Granule.

This paper is an improved version of [4]. The main improvement is the above mentioned formal construction (Section 4), which makes it possible for the programmer to use in the same program different arbitrary coeffect algebras (an example is given in Section 6), without caring about their combination, which is internally handled by the type system. In the preliminary paper, instead, a coeffect class was not really implementing *one* coeffect algebra, since its methods, e.g., the sum, had an argument of a generic `Coeffect` class, and the programmer had to add cases corresponding to coeffect arguments of a different kind. Moreover, in the current paper coeffect classes are introduced as a specific feature which can be added to the Java-like calculus, thus abstracting from language details allowing to implement such feature.

In Section 2 we define a Java-like calculus where variable declarations are annotated with coeffects, taken in an arbitrary coeffect algebra. In Section 3 we provide a type-and-coeffect system for the calculus, parametric on the underlying coeffect algebra, and prove type and coeffects preservation. In Section 4 we show the construction of the coeffect algebra of heterogeneous coeffects. In Section 5 we describe a slight extension of the calculus supporting the declaration of coeffect classes, show the instantiation of the previous parametric type system to the case where coeffects are values of such coeffect classes, provide several examples, and outline an implementation in full Java. In Section 6 we provide a more extended programming example using different kinds of coeffects. We discuss related work in Section 7, and, finally, we summarize the contribution and outline further work in Section 8. The straightforward proof that the construction in Section 4 gives a coeffect algebra is given in the Appendix.

2. Calculus

The calculus which we enrich by *coeffect annotations*, ranged over by r, s, t, is a variant of Featherweight Java [17] (FJ for short), a functional subset of Java which is widely used as reference calculus to study properties and/or propose extensions of Java-like languages.

We assume *variables* $x, y, z, \ldots$ which either are bound in the source code (method parameters, including the special variable `this`, and local variables in blocks) or are free, that is, denote external resources. Moreover, we assume *class names* C, D, *field names* f, and *method names* m, and the standard predefined class `Object`, root of the inheritance hierarchy. We write es as metavariable for $e_1, \ldots, e_n$, $n \geq 0$, and analogously for other sequences.

The syntax of expressions is given in Fig. 1. Standard FJ expressions are variable, field access, constructor invocation, method invocation, and cast (here actually only downcast, see next section). In addition, we include a block expression, relevant for our aims since the variable declaration specifies a coeffect annotation. The format of coeffect annotations is inspired by that used in Granule [19]. Moreover, we add some other features mainly needed to write examples: abstract classes, abstract and static methods, conditional, dynamic typecheck, and booleans with their (omitted) operations. Types are either class names or the predefined primitive type `boolean`.

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

- $\leq$ is the subtyping relation (the reflexive and transitive closure of the `extends` relation)
- $\mathsf{fields}(C)$ gives, for each class C, the sequence of fields with their types, assumed to have all distinct names
- $\mathsf{mbody}(C, m)$ gives, for each method m of class C, the parameters and body.

Reduction rules are given in Fig. 1. Since the language is functional, FJ configurations are expressions, and, in particular, constructor invocations where all arguments are fully evaluated represent *objects* (instances of classes). Indeed, in FJ, each class has exactly one constructor, with a sequence of arguments corresponding to the fields of the class.

Rule (CTX) is the standard contextual rule, where evaluation contexts $\mathcal{E}$ express the usual left-to-right evaluation strategy.

In rule (FIELD-ACCESS), accessing a field of an object succeeds if the field is one of the fields of the object's class. In this case, the field access evaluates to the corresponding value.

Invocation of an instance and static method are modeled by rules (INVK) and (ST-INVK), respectively. The sequence of parameters and the body of the method are retrieved from the class table, and the invocation is reduced to the body where the parameters have been replaced by the corresponding arguments. In the case of an instance method, the implicit `this` parameter is replaced by the receiver as well.

Rules (BLOCK), (IF-TRUE), and (IF-FALSE) are the standard rules for declaration of a local variable and conditional.

Rules (INSTOF-TRUE) and (INSTOF-FALSE) model the dynamic check that an object be an instance of (a subclass of) the specified class. Finally, cast is modeled by rule (CAST). A cast succeeds, hence can be removed, if the object to be reduced is an instance of (a subclass of) the specified class. Otherwise, reduction is stuck (an alternative semantics could raise a dynamic error).

Differently from the original FJ semantics [17], rules are instantiated on *open* expressions, since otherwise the fact that reduction preserves coeffects, in addition to types, would trivially hold. In other words, we model reduction of expressions which refer to external resources. In particular, values are open as well, and a variable can be safely used as constructor or method argument, whereas reduction is stuck when it is used as receiver.

$$\begin{array}{lcll}
e & ::= & x \mid e.f \mid \texttt{new}\ C(es) \mid e.m(es) \mid C.m(es) \mid & \text{expression}\\
& & \{T[r]\ x = e;\ e'\} \mid \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2 \mid & \\
& & e\ \texttt{instanceof}\ C \mid (C)e \mid \texttt{true} \mid \texttt{false} \mid \ldots & \\
T & ::= & C \mid \texttt{boolean} & \text{type}\\
v & ::= & \texttt{new}\ C(vs) \mid x \mid \texttt{true} \mid \texttt{false} & \text{value}\\
\mathcal{E} & ::= & [\,] \mid \mathcal{E}.f \mid \texttt{new}\ C(vs, \mathcal{E}, es) \mid \mathcal{E}.m(es) \mid v.m(vs, \mathcal{E}, es) & \text{evaluation context}\\
& & \mid C.m(vs, \mathcal{E}, es) \mid \{T[r]\ x = \mathcal{E};\ e\} \mid & \\
& & \texttt{if}\ (\mathcal{E})\ e_1\ \texttt{else}\ e_2 \mid \mathcal{E}\ \texttt{instanceof}\ C \mid (C)\mathcal{E} &
\end{array}$$

$$(\text{CTX})\ \frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}$$

$$(\text{FIELD-ACCESS})\ \frac{}{\texttt{new}\ C(v_1, \ldots, v_n).f_i \longrightarrow v_i}\ \begin{array}{l}\mathsf{fields}(C) = T_1\ f_1; \ldots T_n\ f_n;\\ i \in 1..n\end{array}$$

$$(\text{INVK})\ \frac{}{\texttt{new}\ C(vs).m(v_1, \ldots, v_n) \longrightarrow e'}\ \begin{array}{l}\mathsf{mbody}(C, m) = (x_1 \ldots x_n, e)\\ e' = e[\texttt{new}\ C(vs)/\texttt{this}][v_1/x_1 \ldots v_n/x_n]\end{array}$$

$$(\text{ST-INVK})\ \frac{}{C.m(v_1, \ldots, v_n) \longrightarrow e'}\ \begin{array}{l}\mathsf{mbody}(C, m) = (x_1 \ldots x_n, e)\\ e' = e[v_1/x_1] \ldots [v_n/x_n]\end{array}$$

$$(\text{BLOCK})\ \frac{}{\{T[r]\ x = v;\ e\} \longrightarrow e[v/x]}$$

$$(\text{IF-TRUE})\ \frac{}{\texttt{if}\ (\texttt{true})\ e_1\ \texttt{else}\ e_2 \longrightarrow e_1} \qquad (\text{IF-FALSE})\ \frac{}{\texttt{if}\ (\texttt{false})\ e_1\ \texttt{else}\ e_2 \longrightarrow e_2}$$

$$(\text{INSTOF-TRUE})\ \frac{}{\texttt{new}\ D(vs)\ \texttt{instanceof}\ C \longrightarrow \texttt{true}}\ D \leq C$$

$$(\text{INSTOF-FALSE})\ \frac{}{\texttt{new}\ D(vs)\ \texttt{instanceof}\ C \longrightarrow \texttt{false}}\ D \not\leq C$$

$$(\text{CAST})\ \frac{}{(C)\texttt{new}\ D(vs) \longrightarrow \texttt{new}\ D(vs)}\ D \leq C$$

Fig. 1. Calculus.

3. Parametric type-and-coeffect system

In type-and-coeffect systems, the typing judgment has shape $\Gamma \vdash e : T$, where Γ is a (type-and-coeffect) context, that is, a (finite) map from variables to pairs of a coeffect and a type, written $\Gamma = x_1 :_{r_1} T_1, \ldots, x_n :_{r_n} T_n$. We write $\mathsf{dom}(\Gamma)$ for the (finite) domain of Γ. Equivalently, Γ can be seen as the pair of a *coeffect context* and a *type context*, mapping variables to coeffects and types, respectively, with the same (finite) domain. We assume that coeffects form a *coeffect algebra*, specifying *partial order* $\preceq$ with binary *join* $\vee$, *sum* $+$, *multiplication* $\cdot$, *zero coeffect* $\mathbf{0}$, and *one coeffect* $\mathbf{1}$, satisfying some axioms. That is, as detailed in Definition 4.1 in Section 4, they should form a semiring with sum and multiplication monotonic with respect to the partial order, and $\mathbf{0}$ should be the least element.

Our definition is a slight variant of others proposed in literature [8,13,18,2,12,1,19,9,22]. In particular, the partial order models overapproximation in the usage of resources, and allows flexibility, for instance we can have different usage in the branches of an if-then-else construct. The fact that the zero is the least element means that, in particular, overapproximation can add unused variables, making the calculus *affine*.

The typical example of coeffect algebra is the *affinity* algebra, which is used to track whether a variable is unused (0), used at most once (1), or used in an unconstrained way (ω). The partial order and the operations are defined in a pretty intuitive way, as shown below.

$0 \preceq 1 \preceq \omega$

$+$	0	1	ω
0	0	1	ω
1	1	ω	ω
ω	ω	ω	ω

$\cdot$	0	1	ω
0	0	0	0
1	0	1	ω
ω	0	ω	ω

As customary in type-and-coeffect systems, in typing rules contexts are combined by means of some operations, which are, in turn, defined in terms of the corresponding operations on coeffects (grades).

More precisely, we define

Partial order $\emptyset \preceq \emptyset \qquad (x :_r T, \Gamma) \preceq (x :_s T, \Delta)$ if $r \preceq s$ and $\Gamma \preceq \Delta$
$\Gamma \preceq (x :_r T, \Delta)$ if $x \notin \mathsf{dom}(\Gamma)$ and $\Gamma \preceq \Delta$
Binary join $\emptyset \vee \Gamma = \Gamma \qquad (x :_r T, \Gamma) \vee \Delta = x :_r T, (\Gamma \vee \Delta)$ if $x \notin \mathsf{dom}(\Delta)$
$(x :_r T, \Gamma) \vee (x :_s T, \Delta) = x :_{r \vee s} T, (\Gamma \vee \Delta)$
Sum $\emptyset + \Gamma = \Gamma \qquad (x :_r T, \Gamma) + \Delta = x :_r T, (\Gamma + \Delta)$ if $x \notin \mathsf{dom}(\Delta)$
$(x :_r T, \Gamma) + (x :_s T, \Delta) = x :_{r+s} T, (\Gamma + \Delta)$
Scalar multiplication $r \cdot \emptyset = \emptyset \qquad r \cdot (x :_s T, \Gamma) = x :_{r \cdot s} T, (r \cdot \Gamma)$

As the reader may notice, these operations on type-and-coeffect contexts can be equivalently defined by lifting the corresponding operations on coeffect contexts, which are the pointwise extension of those on coeffects, to handle types as well. In this step, the sum and the join operators becomes partial since a variable in the domain of both contexts is required to have the same type.

In the following, we assume the standard precedence of multiplication over sum.

The type-and-coeffect system for the calculus introduced in the previous section relies on the type information extracted from the class table, which, again to be concise, is abstractly modeled as follows:

- $\neg\mathsf{abs}(C)$ means that C is a non-abstract class
- $\mathsf{mtype}(C, m)$ gives, for each method m of class C, its enriched method type, including coeffect annotations, that is, of shape:
  - either $r_0, T_1^{r_1} \ldots T_n^{r_n} \to T$
  - or $T_1^{r_1} \ldots T_n^{r_n} \to T$, meaning that the method is static.

In a well-typed class table, we expect the following conditions to hold:

(T-METH) $\mathsf{mtype}(C, m) = r_0, T_1^{r_1} \ldots T_n^{r_n} \to T$ and $\neg\mathsf{abs}(C)$ implies
$\mathsf{mbody}(C, m) = (x_1 \ldots x_n, e)$ and
$\mathtt{this} :_{r_0} C, x_1 :_{r_1} T_1, \ldots, x_n :_{r_n} T_n \vdash e : T$
(T-ST-METH) $\mathsf{mtype}(C, m) = T_1^{r_1} \ldots T_n^{r_n} \to T$ implies
$\mathsf{mbody}(C, m) = (x_1 \ldots x_n, e)$ and
$x_1 :_{r_1} T_1, \ldots, x_n :_{r_n} T_n \vdash e : T$
(T-INH-FIELDS) $C \leq D$ implies $\mathsf{fields}(D)$ is a prefix of $\mathsf{fields}(C)$
(T-INH-METH) $C \leq D$ and $\mathsf{mtype}(D, m) = r_0, T_1^{r_1} \ldots T_n^{r_n} \to T$ imply
$\mathsf{mtype}(C, m) = s_0, T_1^{s_1} \ldots T_n^{s_n} \to T'$
with $T' \leq T$, $s_i \preceq r_i$ for $i \in 0..n$

Conditions (T-METH) and (T-ST-METH) express that method bodies should conform to method types. Condition (T-INH-FIELDS) expresses that fields are inherited, and, together with the assumption that they have distinct names, that there is no field hiding. Finally, condition (T-INH-METH) expresses that methods are inherited, cannot be overloaded, and can be overriden with a more specific return type, and the more restrictive coeffects. Note that this condition only concerns instance methods, indeed static methods are not inherited.

In Fig. 2, we describe the typing rules, which are *parameterized* on the underlying coeffect algebra.

In the subsumption rule (T-SUB), both the coeffect context and the type can be made more general. This means that variables can get less constraining coeffects. For instance, assuming again affinity coeffects, an expression which can be typechecked assuming to use a given variable at most once (coeffect 1) can be typechecked with no constraints (coeffect ω).

In rule (T-VAR), the given variable is used exactly once, and no other variable is used. In rules (T-FIELD-ACCESS) and (T-NEW), coeffects of the subterms are summed.

In rule (T-INVK), the coeffects of the arguments are summed, after multiplying each of them with the join (least upper bound) of the coeffect annotation of the corresponding parameter, and the one coeffect. This guarantees to take into account the coeffects of the initialization expression for parameters not used in the body, as needed in type-and-coeffect systems for call-by-value calculi (see the end of Example 3.1 below). The rule uses the auxiliary function mtype mentioned before, which returns an enriched method type, where the types of the parameters and of `this` have coeffect annotations. Rule (T-ST-INVK) is the analogous rule for static methods.

In rule (T-BLOCK), the coeffects of the initialization expression are multiplied by the join of the coeffect annotation of the variable, and the one coeffect, and then summed with those of the body. Analogously to method invocation, the join with the one coeffect is needed when the variable is not used in the body. Note that the variable is used in the body accordingly with the annotation written by the programmer.

$$\text{(T-SUB)}\ \frac{\Gamma \vdash e : T \quad \Gamma \preceq \Gamma'}{\Gamma' \vdash e : T' \quad T \leq T'} \qquad \text{(T-VAR)}\ \frac{}{x :_{\mathbf{1}} T \vdash x : T}$$

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

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

$$\text{(T-INVK)}\ \frac{\Gamma_0 \vdash e_0 : C \quad \Gamma_i \vdash e_i : T_i \quad \forall i \in 1..n}{s_0 \cdot \Gamma_0 + \ldots + s_n \cdot \Gamma_n \vdash e_0.m(e_1, \ldots, e_n) : T} \quad \begin{array}{l}\mathsf{mtype}(C, m) = r_0, T_1^{r_1} \ldots T_n^{r_n} \to T\\ s_i = r_i \vee \mathbf{1} \quad \forall i \in 0..n\end{array}$$

$$\text{(T-ST-INVK)}\ \frac{\Gamma_i \vdash e_i : T_i \quad \forall i \in 1..n}{s_1 \cdot \Gamma_1 + \ldots + s_n \cdot \Gamma_n \vdash C.m(e_1, \ldots, e_n) : T} \quad \begin{array}{l}\mathsf{mtype}(C, m) = T_1^{r_1} \ldots T_n^{r_n} \to T\\ s_i = r_i \vee \mathbf{1} \quad \forall i \in 1..n\end{array}$$

$$\text{(T-BLOCK)}\ \frac{\Gamma \vdash e : T \qquad \Gamma', x :_r T \vdash e' : T'}{s \cdot \Gamma + \Gamma' \vdash \{T[r]\ x = e;\ e'\} : T'} \quad s = r \vee \mathbf{1}$$

$$\text{(T-IF)}\ \frac{\Gamma \vdash e : \mathtt{boolean} \quad \Delta_1 \vdash e_1 : T \quad \Delta_2 \vdash e_2 : T}{\Gamma + (\Delta_1 \vee \Delta_2) \vdash \mathtt{if}\ (e)\ e_1\ \mathtt{else}\ e_2 : T}$$

$$\text{(T-INSTOF)}\ \frac{\Gamma \vdash e : D}{\Gamma \vdash e\ \mathtt{instanceof}\ C : \mathtt{boolean}} \qquad \text{(T-CAST)}\ \frac{\Gamma \vdash e : D}{\Gamma \vdash (C)e : C} \quad C \leq D$$

$$\text{(T-TRUE)}\ \frac{}{\emptyset \vdash \mathtt{true} : \mathtt{boolean}} \qquad \text{(T-FALSE)}\ \frac{}{\emptyset \vdash \mathtt{false} : \mathtt{boolean}}$$

Fig. 2. Parametric type-and-coeffect system.

In rule (T-IF), the join operator is applied to the contexts of the two branches. The result is a context where each variable has a coeffect which is greater (less constraining) than those in the two branches. This guarantees that, regardless of which branch will be executed, each variable will have the right amount of resources. Then, the coeffects of this context are summed with those in the context of the guard. Note that we could have equivalently given a rule where the same context is imposed for the two branches, since this can be obtained by subsumption; however, in the instantiation in Section 5, this more effective version of the rule corresponds to the fact that the join context is computed through a user-defined method.

Rules (T-INSTOF), (T-CAST), (T-TRUE) and (T-FALSE) are straightforward, apart that we only allow downcast. This is just to avoid the well-known (orthogonal) problem [17] that subject reduction is not preserved by allowing upcast as well. Also note that, as in the original FJ paper, the standard formulation of progress does not hold, since failure of a downcast is for simplicity modeled by a stuck computation. This is not an issue, since here we are only interested in subject reduction.

Example 3.1. We illustrate the use of the type-and-coeffect system on a simple class table, assuming the affinity coeffects 0 (unused), 1 (used at most once), ω (no constraints) introduced before. Here they occur as annotation of `this`, written between the method name and the list of parameters. In the examples, for brevity, we omit the stylized constructor and `extends Object`, required in the original FJ paper [17].

```
class Pair{A fst; A snd;}
class A{
  A drop [0] () {new A()}
  A identity [1] () {this}
  Pair duplicate [ω] () { new Pair(this,this)}
}
```

Let us see some examples of how the type system works. The above declarations correspond to have:

$\mathsf{mtype}(\mathtt{A}, \mathtt{drop}) = 0, \epsilon \to \mathtt{A}$
$\mathsf{mtype}(\mathtt{A}, \mathtt{identity}) = 1, \epsilon \to \mathtt{A}$
$\mathsf{mtype}(\mathtt{A}, \mathtt{duplicate}) = \omega, \epsilon \to \mathtt{Pair}$

To check that, e.g., the method `duplicate` is well-typed, we have to typecheck the method body, and then verify the condition (T-METH) at page 4. A type derivation for the method body is as follows:

$$\dfrac{\dfrac{}{\texttt{this} :_1 \texttt{A} \vdash \texttt{this} : \texttt{A}}\,(\textsc{t-var}) \qquad \dfrac{}{\texttt{this} :_1 \texttt{A} \vdash \texttt{this} : \texttt{A}}\,(\textsc{t-var})}{\texttt{this} :_\omega \texttt{A} \vdash \texttt{new}\ \texttt{Pair(this, this)} : \texttt{Pair}}\,(\textsc{t-new})$$

so the condition of rule (t-meth) holds. Analogously we can see that the other two methods are well-typed.

To see an example of ill-typed method, assume, e.g., that the expression `new Pair(this, this).fst` was the body of method `identity`. Indeed, we would have a similar derivation:

$$\dfrac{\dfrac{\dfrac{}{\texttt{this} :_1 \texttt{A} \vdash \texttt{this} : \texttt{A}}\,(\textsc{t-var}) \qquad \dfrac{}{\texttt{this} :_1 \texttt{A} \vdash \texttt{this} : \texttt{A}}\,(\textsc{t-var})}{\texttt{this} :_\omega \texttt{A} \vdash \texttt{new}\ \texttt{Pair(this, this)} : \texttt{Pair}}\,(\textsc{t-new})}{\texttt{this} :_\omega \texttt{A} \vdash \texttt{new}\ \texttt{Pair(this, this).fst} : \texttt{A}}\,(\textsc{t-field-access})$$

However, $\texttt{this} :_\omega \texttt{A} \preceq \texttt{this} :_1 \texttt{A}$ does not hold, so we cannot apply rule (t-sub), and condition (t-meth) does not hold.

A call of `duplicate` can be typed as shown below:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{x} :_1 \texttt{A} \vdash \texttt{x} : \texttt{A}}\,(\textsc{t-var}) \qquad \dfrac{}{\texttt{y} :_1 \texttt{A} \vdash \texttt{y} : \texttt{A}}\,(\textsc{t-var})}{\texttt{x} :_1 \texttt{A}, \texttt{y} :_1 \texttt{A} \vdash \texttt{new}\ \texttt{Pair(x, y)} : \texttt{Pair}}\,(\textsc{t-new})}{\texttt{x} :_1 \texttt{A}, \texttt{y} :_1 \texttt{A} \vdash \texttt{new}\ \texttt{Pair(x, y).fst} : \texttt{A}}\,(\textsc{t-field-access})}{\texttt{x} :_\omega \texttt{A}, \texttt{y} :_\omega \texttt{A} \vdash \texttt{new}\ \texttt{Pair(x, y).fst.duplicate()} : \texttt{Pair}}\,(\textsc{t-invk})$$

since $\mathsf{mtype}(\texttt{A}, \texttt{duplicate}) = \omega, \epsilon \to \texttt{Pair}$, $\omega = \omega \vee 1$, and $\omega = \omega \cdot 1$.

Finally, we show an example motivating the need for the join with 1 in rules (t-invk), (t-st-invk), and (t-block). For instance, this prevents to derive the judgment $\texttt{y} :_0 \texttt{A} \vdash \{\texttt{Pair[0] x = y.duplicate; new A()}\} : \texttt{A}$, incorrectly stating that the variable `y` is not used, whereas it is used in the initialization expression of `x`. The join with 1 would be not needed in a call-by-name calculus.

Our main technical result is subject reduction (Theorem 3.5), expressing, as customary in type-and-coeffect systems, that not only the type, but also the coeffects are preserved by reduction. By subsumption, this means that the type can become more specific, and the coeffects more constraining, as illustrated by the example below:

$e = \texttt{if (true) new Pair(x, new A()) else (Object)new Pair(x, x)}$
$\texttt{x} :_\omega \texttt{A} \vdash e : \texttt{Object}$
$e \longrightarrow e' = \texttt{new Pair(x, new A())}$
$\texttt{x} :_1 \texttt{A} \vdash e' : \texttt{A}$

The proof of Theorem 3.5 uses the standard lemmas of inversion for expressions and contexts (Lemmas 3.2 and 3.3), and substitution (Lemma 3.4).

Lemma 3.2 *(Inversion).*

1. *If $\Gamma \vdash x : T$, then $x :_{\mathbf{1}} T' \preceq \Gamma$ with $T' \leq T$.*
2. *If $\Gamma \vdash e.f_i : T$, then $\Gamma' \vdash e : C$, and $\mathsf{fields}(C) = T_1\, f_1; \ldots T_n\, f_n;$, $i \in 1..n$, with $\Gamma' \preceq \Gamma$ and $T_i \leq T$.*
3. *If $\Gamma \vdash \texttt{new}\ C(e_1, \ldots, e_n) : T$, then $\Gamma_i \vdash e_i : T_i$ for all $i \in 1..n$, $\neg\mathsf{abs}(C)$, and $\mathsf{fields}(C) = T_1\, f_1; \ldots T_n\, f_n;$, with $\Gamma_1 + \ldots + \Gamma_n \preceq \Gamma$ and $C \leq T$.*
4. *If $\Gamma \vdash e_0.m(e_1, \ldots, e_n) : T$, then $\Gamma_0 \vdash e_0 : C$, and $\Gamma_i \vdash e_i : T_i$ for all $i \in 1..n$, and $\mathsf{mtype}(C, m) = r_0, T_1^{r_1} \ldots T_n^{r_n} \to T'$, with $s_0 \cdot \Gamma_0 + \ldots + s_n \cdot \Gamma_n \preceq \Gamma$, where $s_i = r_i \vee \mathbf{1}$ for all $i \in 0..n$, and $T' \leq T$.*
5. *If $\Gamma \vdash C.m(e_1, \ldots, e_n) : T$, then $\Gamma_i \vdash e_i : T_i$ for all $i \in 1..n$, and $\mathsf{mtype}(C, m) = T_1^{r_1} \ldots T_n^{r_n} \to T'$, with $s_1 \cdot \Gamma_1 + \ldots + s_n \cdot \Gamma_n \preceq \Gamma$, where $s_i = r_i \vee \mathbf{1}$ for all $i \in 1..n$, and $T' \leq T$.*
6. *If $\Gamma \vdash \{T_1[r]\, x = e_1;\ e_2\} : T$, then $\Gamma_1 \vdash e_1 : T_1$ and $\Gamma_2, x :_r T_1 \vdash e_2 : T_2$, with $s \cdot \Gamma_1 + \Gamma_2 \preceq \Gamma$, where $s = r \vee \mathbf{1}$, and $T_2 \leq T$.*
7. *If $\Gamma \vdash \texttt{if}\,(e)\, e_1\ \texttt{else}\ e_2 : T$, then $\Gamma' \vdash e : \texttt{boolean}$, $\Delta_1 \vdash e_1 : T'$ and $\Delta_2 \vdash e_2 : T'$, with $\Gamma' + (\Delta_1 \vee \Delta_2) \preceq \Gamma$ and $T' \leq T$.*
8. *If $\Gamma \vdash e\ \texttt{instanceof}\ C : T$, then $T = \texttt{boolean}$ and $\Gamma' \vdash e : D$, with $\Gamma' \preceq \Gamma$.*
9. *If $\Gamma \vdash (C)e : T$, then $\Gamma' \vdash e : D$, and $C \leq D$, with $C \leq T$, and $\Gamma' \preceq \Gamma$.*

Proof. By cases on typing rules. □

Lemma 3.3 *(Context Inversion). If $\Gamma \vdash \mathcal{E}[e] : T$, then $\Gamma', x :_r T' \vdash \mathcal{E}[x] : T$ and $\Delta \vdash e : T'$ for some Γ', Δ, $x \notin \mathsf{dom}(\Gamma)$, r and T' such that $\Gamma' + r \cdot \Delta \preceq \Gamma$.*

Proof. By straightforward induction on the structure of $\mathcal{E}$ using Lemma 3.2. □

Lemma 3.4 *(Substitution). If $\Delta \vdash e' : T'$ and $\Gamma, x :_r T' \vdash e : T$ then $\Gamma + r \cdot \Delta \vdash e[e'/x] : T$.*

Proof. By straightforward induction on the derivation of $\Gamma, x :_r T' \vdash e : T$. □

Theorem 3.5. *If $\Gamma \vdash e : T$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : T$.*

Proof. We proceed by induction on reduction rules.

(ctx) We have $\mathcal{E}[e] \longrightarrow \mathcal{E}[e']$ and $e \longrightarrow e'$. From $\Gamma \vdash \mathcal{E}[e] : T$, by Lemma 3.3, we have $\Gamma', x :_r T' \vdash \mathcal{E}[x] : T$ and $\Delta \vdash e : T'$ with $x \notin \mathsf{dom}(\Gamma)$ and $\Gamma' + r \cdot \Delta \preceq \Gamma$. By induction hypothesis we derive $\Delta \vdash e' : T'$. Then, by Lemma 3.4, we get $\Gamma' + r \cdot \Delta \vdash \mathcal{E}[e'] : T$ since $\mathcal{E}[x][e'/x] = \mathcal{E}[e']$. Finally, the thesis follows by rule (t-sub).

(field-access) We have $\texttt{new}\ C(v_1, \ldots, v_n).f_i \longrightarrow v_i$ and $\mathsf{fields}(C) = T_1\ f_1; \ldots T_n\ f_n;$ with $i \in 1..n$. From $\Gamma \vdash \texttt{new}\ C(v_1, \ldots, v_n).f_i : T$, by Lemma 3.2(2), we have $\Gamma' \vdash \texttt{new}\ C(v_1, \ldots, v_n) : D$ and $\mathsf{fields}(D) = T'_1\ f'_1; \ldots T'_m\ f'_m;$ and $f_i = f'_k$, for some $k \in 1..m$, with $\Gamma' \preceq \Gamma$ and $T'_k \le T$. By Lemma 3.2(3), we know that $C \le D$ and $\Gamma_j \vdash v_j : T_j$ for all $j \in 1..n$, with $\Gamma_1 + \ldots + \Gamma_n \preceq \Gamma'$. By condition (t-inh-fields), we get $m \le n$ and $T'_j\ f'_j; = T_j\ f_j;$, for all $j \in 1..m$, hence, in particular, $i = k \le m$ and $T_i = T'_k$. Therefore, we have $\Gamma_i \vdash v_i : T_i$ and, since $T_i = T'_k \le T$ and $\Gamma_i \preceq \Gamma_1 + \ldots + \Gamma_n \preceq \Gamma' \preceq \Gamma$, we get the thesis by rule (t-sub).

(invk) We have $\texttt{new}\ C(vs).m(v_1, \ldots, v_n) \longrightarrow e[\texttt{new}\ C(vs)/\texttt{this}][v_1/x_1 \ldots v_n/x_n]$ and $\mathsf{mbody}(C, m) = (x_1 \ldots x_n, e)$. From $\Gamma \vdash \texttt{new}\ C(vs).m(v_1, \ldots, v_n) : T$, by Lemma 3.2(4), we have $\Gamma_0 \vdash \texttt{new}\ C(vs) : D$ and $\Gamma_i \vdash v_i : T_i$ for all $i \in 1..n$, and $\mathsf{mtype}(D, m) = r_0, T_1^{r_1} \ldots T_n^{r_n} \to T'$, with $s_0 \cdot \Gamma_0 + \ldots + s_n \cdot \Gamma_n \preceq \Gamma$, where $s_i = r_i \vee \mathbf{1}$ for all $i \in 0..n$, and $T' \le T$. By Lemma 3.2(3), we have $C \le D$ and $\neg\mathsf{abs}(C)$, hence, by (t-inh-meth), we get $\mathsf{mtype}(C, m) = t_0, T_1^{t_1} \ldots T_n^{t_n} \to T''$ with $T'' \le T'$ and $t_i \preceq r_i$ for $i \in 0..n$. Then, by condition (t-meth), we also get $\texttt{this} :_{t_0} C, x_1 :_{t_1} T_1, \ldots, x_n :_{t_n} T_n \vdash e : T''$. By iteratively applying Lemma 3.4 to all variables $\texttt{this}, x_1, \ldots, x_n$, we get $t_0 \cdot \Gamma_0 + \ldots + t_n \cdot \Gamma_n \vdash e[\texttt{new}\ C(vs)/\texttt{this}][v_1/x_1 \ldots v_n/x_n] : T''$. Since $t_i \preceq r_i \preceq s_i$ for all $i \in 0..n$, we have $t_0 \cdot \Gamma_0 + \ldots + t_n \cdot \Gamma_n \preceq s_0 \cdot \Gamma_0 + \ldots + s_n \cdot \Gamma_n \preceq \Gamma$ and $T'' \le T' \le T$. Then, by rule (t-sub), we get the thesis.

(st-invk) Analogous to (invk).

(block) We have $\{T_1[r]\ x = v;\ e'\} \longrightarrow e'[v/x]$. From $\Gamma \vdash \{T_1[r]\ x = v;\ e'\} : T$, by Lemma 3.2(6), we have $\Gamma_1 \vdash v : T_1$, and $\Gamma_2, x :_r T_1 \vdash e' : T'$, with $s \cdot \Gamma_1 + \Gamma_2 \preceq \Gamma$, where $s = r \vee \mathbf{1}$, and $T' \le T$. By Lemma 3.4 and by (t-sub), we derive $s \cdot \Gamma_1 + \Gamma_2 \vdash e'[v/x] : T'$. Since $s \cdot \Gamma_1 + \Gamma_2 \preceq \Gamma$ and $T' \le T$, by rule (t-sub) we get the thesis.

(if-true) We have $\texttt{if}\,(\texttt{true})\,e_1\,\texttt{else}\ e_2 \longrightarrow e_1$. From $\Gamma \vdash \texttt{if}\,(\texttt{true})\,e_1\,\texttt{else}\ e_2 : T$, by Lemma 3.2(7), we have $\Gamma' \vdash \texttt{true} : \texttt{boolean}$, $\Delta_1 \vdash e_1 : T'$ and $\Delta_2 \vdash e_2 : T'$, with $\Gamma' + (\Delta_1 \vee \Delta_2) \preceq \Gamma$ and $T' \le T$. We have $\Delta_1 \preceq \Delta_1 \vee \Delta_2$, so, since $\Delta_1 \preceq \Delta_1 \vee \Delta_2 \preceq \Gamma' + (\Delta_1 \vee \Delta_2) \preceq \Gamma$ and $T' \le T$, by rule (t-sub) we derive $\Gamma \vdash e_1 : T$.

(if-false) Analogous to (if-true).

(instof-true) We have $\texttt{new}\ C'(e_1, \ldots, e_n)\ \texttt{instanceof}\ C \longrightarrow \texttt{true}$ with $C' \le C$. From $\Gamma \vdash \texttt{new}\ C'(e_1, \ldots, e_n)\ \texttt{instanceof}\ C : T$, by Lemma 3.2(8), we have $T = \texttt{boolean}$ and $\Gamma' \vdash \texttt{new}\ C'(e_1, \ldots, e_n) : D$, with $\Gamma' \preceq \Gamma$. By (t-true) we have $\emptyset \vdash \texttt{true} : T$. Since $\emptyset \preceq \Gamma$, by rule (t-sub) we derive $\Gamma \vdash \texttt{true} : T$.

(instof-false) Analogous to (instof-true).

(cast) We have $(C)\texttt{new}\ C'(v_1, \ldots, v_n) \longrightarrow \texttt{new}\ C'(v_1, \ldots, v_n)$ with $C' \le C$. From $\Gamma \vdash (C)\texttt{new}\ C'(v_1, \ldots, v_n) : T$, by Lemma 3.2(9), we have $\Gamma' \vdash \texttt{new}\ C'(v_1, \ldots, v_n) : D$, and $C \le D$, with $\Gamma' \preceq \Gamma$ and $C \le T$. By Lemma 3.2(3) we have $\Gamma_i \vdash v_i : T_i$ for all $i \in 1..n$, $\neg\mathsf{abs}(C')$, and $\mathsf{fields}(C') = T_1\ f_1; \ldots T_n\ f_n;$, with $\Gamma_1 + \ldots + \Gamma_n \preceq \Gamma'$ and $C' \le D$. By rule (t-new) we derive $\Gamma_1 + \ldots + \Gamma_n \vdash \texttt{new}\ C'(v_1, \ldots, v_n) : C'$. Since $\Gamma_1 + \ldots + \Gamma_n \preceq \Gamma' \preceq \Gamma$ and $C' \le C \le T$, by rule (t-sub) we derive $\Gamma \vdash \texttt{new}\ C'(v_1, \ldots, v_n) : T$. □

4. Combining coeffect algebras

We formally define coeffect algebras and related notions, and a construction which, given a family of coeffect algebras, returns a unique coeffect algebra of heterogeneous coeffects.

Definition 4.1 *(Coeffect algebra).* A *coeffect algebra* is a tuple $\mathcal{R} = \langle |\mathcal{R}|, \preceq, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$ such that:

- $\langle |\mathcal{R}|, \preceq \rangle$ is a partially ordered set, with binary joins $\vee$;
- $\langle |\mathcal{R}|, \preceq, +, \mathbf{0} \rangle$ is a partially ordered commutative monoid;
- $\langle |\mathcal{R}|, \preceq, \cdot, \mathbf{1} \rangle$ is a partially ordered monoid;

and, moreover, the following axioms are satisfied:

- $r \cdot (s + t) = r \cdot s + r \cdot t$ and $(s + t) \cdot r = s \cdot r + t \cdot r$, for all $r, s, t \in |\mathcal{R}|$;
- $r \cdot \mathbf{0} = \mathbf{0}$ and $\mathbf{0} \cdot r = \mathbf{0}$, for all $r \in |\mathcal{R}|$;

- $\mathbf{0} \preceq r$, for all $r \in |\mathcal{R}|$.

Essentially, a coeffect algebra is a partially ordered semiring, that is, a semiring together with a partial order relation on its underlying set, making addition and multiplication monotonic with respect to it and having the zero as its least element. The partial order relation is assumed to have binary joins; that is, for any pair of coeffects r, s, there is a coeffect, denoted $r \vee s$, such that $r \preceq (r \vee s)$ and $s \preceq (r \vee s)$, that is, it is an upper bound for r and s, and, for each other upper bound t, $(r \vee s) \preceq t$, that is, it is the *least* upper bound (join). Without this property, typing rules (T-INVK), (T-ST-INVK), (T-BLOCK), and (T-IF) in Fig. 2 should be expressed in a non-algorithmic way, relying on the existence of some upper bound. A homomorphism of coeffect algebras $f : \mathcal{R} \to S$ is a monotone function $f : \langle|\mathcal{R}|, \preceq_{\mathcal{R}}\rangle \to \langle|S|, \preceq_S\rangle$ between the underlying partial orders, which preserves binary joins and the semiring structure, that is, it has to satisfy the following equations:

- $f(r \vee_{\mathcal{R}} s) = f(r) \vee_S f(s)$, for all $r, s \in |\mathcal{R}|$;
- $f(\mathbf{0}_{\mathcal{R}}) = \mathbf{0}_S$ and $f(r +_{\mathcal{R}} s) = f(r) +_S f(s)$, for all $r, s \in |\mathcal{R}|$;
- $f(\mathbf{1}_{\mathcal{R}}) = \mathbf{1}_S$ and $f(r \cdot_{\mathcal{R}} s) = f(r) \cdot_S f(s)$, for all $r, s \in |\mathcal{R}|$.

Coeffect algebras and their homomorphisms form a category denoted by $\mathcal{C}oeff\mathcal{A}lg$.

The following example presents two coeffect algebras which will play an important role.

Example 4.2.

1. The semiring $\mathcal{N} = \langle \mathbb{N}, \leq, +, \cdot, 0, 1\rangle$ with the natural order and the usual arithmetic operations is a coeffect algebra.
2. The trivial semiring $\mathcal{T}$, whose carrier is a singleton set $|\mathcal{T}| = \{\infty\}$, the partial order is the equality, addition and multiplication are defined in the trivial way and $\mathbf{0}_{\mathcal{T}} = \mathbf{1}_{\mathcal{T}} = \infty$, is a coeffect algebra.

It is easy to see that, given a coeffect algebra $\mathcal{R}$, if $\mathbf{0} = \mathbf{1}$, then $\mathcal{R}$ is isomorphic to $\mathcal{T}$. Indeed, for all $r \in |\mathcal{R}|$, we have $r = \mathbf{1} \cdot r = \mathbf{0} \cdot r = \mathbf{0}$, hence the underlying set of $\mathcal{R}$ is a singleton and so it is isomorphic to $\mathcal{T}$.

Consider a coeffect algebra $\mathcal{R}$. Then, we can define functions $\iota_{\mathcal{R}} : |\mathcal{N}| \to |\mathcal{R}|$ and $\zeta_{\mathcal{R}} : |\mathcal{R}| \to |\mathcal{T}|$ as follows:

$$\iota_{\mathcal{R}}(n) = \begin{cases} \mathbf{0}_{\mathcal{R}} & \text{if } n = 0 \\ \iota_{\mathcal{R}}(m) +_{\mathcal{R}} \mathbf{1}_{\mathcal{R}} & \text{if } n = m+1 \end{cases} \qquad \zeta_{\mathcal{R}}(r) = \infty$$

That is, $\iota_{\mathcal{R}}$ maps a natural number n to the sum in $\mathcal{R}$ of n copies of $\mathbf{1}_{\mathcal{R}}$, while $\zeta_{\mathcal{R}}$ maps every element of $\mathcal{R}$ to ∞. Both these functions give rise to homomorphisms $\zeta_{\mathcal{R}} : \mathcal{R} \to \mathcal{T}$ and $\iota_{\mathcal{R}} : \mathcal{N} \to \mathcal{R}$. This fact for $\zeta_{\mathcal{R}}$ is straightforward, and for $\iota_{\mathcal{R}}$ is proved in Proposition 4.3 below. Moreover, $\iota_{\mathcal{R}}$ is the *unique* homomorphism from $\mathcal{N}$ to $\mathcal{R}$, and, conversely, $\zeta_{\mathcal{R}}$ is the *unique* homomorphism from $\mathcal{R}$ to $\mathcal{T}$. In other words, in the terminology of category theory, $\mathcal{N}$ and $\mathcal{T}$ are, respectively, the initial and final object in the category $\mathcal{C}oeff\mathcal{A}lg$ of coeffect algebras with their homomorphisms. This property is important in the construction of a unique coeffect algebra of heterogeneous coeffects from a family of coeffect algebras, as described in the following.

Proposition 4.3. *The following facts hold:*

1. *$\mathcal{N}$ is the initial object in $\mathcal{C}oeff\mathcal{A}lg$;*
2. *$\mathcal{T}$ is the terminal object in $\mathcal{C}oeff\mathcal{A}lg$.*

Proof. Item 2 is straightforward as the singleton set is a terminal object in the category of sets and functions. Towards a proof of Item 1, let $f : \mathcal{N} \to \mathcal{R}$ be a coeffect algebra homomorphism and note that, since $n = 1 + \cdots + 1$ (n times), for all $n \in \mathbb{N}$, and f preserves sums and the unit, we get $f(n) = f(1) +_{\mathcal{R}} \cdots +_{\mathcal{R}} f(1) = \mathbf{1}_{\mathcal{R}} +_{\mathcal{R}} \cdots +_{\mathcal{R}} \mathbf{1}_{\mathcal{R}}$ (n times). That is, we have $f(n) = \iota_{\mathcal{R}}(n)$, for all $n \in \mathbb{N}$. Therefore, to conclude, we just have to show that the map $\iota_{\mathcal{R}}$ is a coeffect algebra homomorphism. The fact that $\iota_{\mathcal{R}}(0) = \mathbf{0}_{\mathcal{R}}$ and $\iota_{\mathcal{R}}(1) = \mathbf{1}_{\mathcal{R}}$ is immediate. The fact that $\iota_{\mathcal{R}}(n+m) = \iota_{\mathcal{R}}(n) +_{\mathcal{R}} \iota_{\mathcal{R}}(m)$ and $\iota_{\mathcal{R}}(n \cdot m) = \iota_{\mathcal{R}}(n) \cdot_{\mathcal{R}} \iota_{\mathcal{R}}(m)$ follows from a straightforward induction on n, using distributivity and nullity properties of the coeffect algebra $\mathcal{R}$. In order to prove monotonicity, consider $n \leq m$ and proceed by induction on $m - n$. If $m - n = 0$, then $n = m$ and so the thesis is trivial. If $m - n = k+1$, we have $m - (n+1) = k$, then by induction hypothesis we get $\iota_{\mathcal{R}}(n+1) \preceq_{\mathcal{R}} \iota_{\mathcal{R}}(m)$. Since $\iota_{\mathcal{R}}(n+1) = \iota_{\mathcal{R}}(n) +_{\mathcal{R}} \iota_{\mathcal{R}}(1)$ and $\mathbf{0}_{\mathcal{R}} \preceq_{\mathcal{R}} \iota_{\mathcal{R}}(1)$, we get $\iota_{\mathcal{R}}(n) = \iota_{\mathcal{R}}(n) +_{\mathcal{R}} \mathbf{0}_{\mathcal{R}} \preceq_{\mathcal{R}} \iota_{\mathcal{R}}(n) +_{\mathcal{R}} \iota_{\mathcal{R}}(1) \preceq_{\mathcal{R}} \iota_{\mathcal{R}}(m)$, as needed. Finally, to prove that $\iota_{\mathcal{R}}$ preserves binary joins, note that $n \vee m$ in $\mathcal{N}$ is either n or m as either $n \leq m$ or $m \leq n$. Let us assume $n \leq m$ hence $n \vee m = m$, the other case is analogous. By monotonicity of $\iota_{\mathcal{R}}$, we have $\iota_{\mathcal{R}}(n) \preceq_{\mathcal{R}} \iota_{\mathcal{R}}(m)$, hence $\iota_{\mathcal{R}}(n) \vee_{\mathcal{R}} \iota_{\mathcal{R}}(m) = \iota_{\mathcal{R}}(m) = \iota_{\mathcal{R}}(n \vee m)$, as needed. □

We describe now a construction which, given a family of coeffect algebras, returns a unique coeffect algebra of *heterogeneous coeffects*. The fact that the construction actually gives a coeffect algebra is modularly expressed by some lemmas and a main theorem; all proofs are in the Appendix.

In the following, we assume a set of *kinds* K including Nat and Triv and a K-indexed family of coeffect algebras $(\mathcal{R}_k)_{k\in K}$ such that $\mathcal{R}_{\mathsf{Nat}} = \mathcal{N}$ and $\mathcal{R}_{\mathsf{Triv}} = \mathcal{T}$. For each $k \in K$, we abbreviate by ι_k and ζ_k, respectively, the homorphisms $\iota_{\mathcal{R}_k}$ and $\zeta_{\mathcal{R}_k}$ defined above.

Heterogeneous coeffects The set of heterogeneous coeffects is defined as $|\mathcal{H}| = \{k{:}r \mid r \in |\mathcal{R}_k|\}$. That is, they are all those of the coeffect algebras in the family, each one paired with its original kind.

Coeffects counting occurrences (natural numbers) and the trivial coeffect are assumed to be always included for the following reasons. The $\mathbf{1}$ in the coeffect algebra of heterogeneous coffects will be $\mathsf{Nat}{:}1$, that is, that of natural numbers. Such coeffect is assigned to any occurrence of a variable, see rule (T-VAR) in Fig. 2. This means that bottom-up computations of coeffects always start by counting occurrences; when a coeffect needs to be combined with another, this is always possible since natural numbers can be mapped into coeffects of any kind, with the ι_k homomorphism. On the other hand, apart from natural numbers, the result of combining coeffects of different kinds will always be the trivial coeffect.

Partial order The partial order $\preceq_{\mathcal{H}}$ on $|\mathcal{H}|$ is defined as follows:

$(\preceq_{\mathcal{H}} 1)$	$k{:}r \preceq_{\mathcal{H}} k{:}s$	iff $r \preceq_k s$, $k \neq \mathsf{Triv}$
$(\preceq_{\mathcal{H}} 2)$	$k{:}r \preceq_{\mathcal{H}} \mathsf{Triv}{:}\infty$	for all k and r
$(\preceq_{\mathcal{H}} 3)$	$\mathsf{Nat}{:}n \preceq_{\mathcal{H}} k{:}s$	iff $\iota_k(n) \preceq_k s$, $k \neq \mathsf{Nat}, \mathsf{Triv}$

where $\preceq_k$ is the partial order of algebra $\mathcal{R}_k$. Here and in the following, we emphasize in grey conditions which are only added to have non-overlapping cases, otherwise we should prove well-definedness.

The partial order on coeffects of the same kind is preserved; coeffects of different kinds are uncomparable, with two exceptions: the trivial coeffect is an upper bound of any other, and a natural number is a lower bound of a coeffect of a certain kind if the same holds for its image in such kind, obtained through the unique homomorphism.

Lemma 4.4. *$\langle|\mathcal{H}|, \preceq_{\mathcal{H}}\rangle$ is a partially ordered set.*

We define the binary join operator $\vee_{\mathcal{H}}$ as follows:

$(\vee_{\mathcal{H}}1)$	$k{:}r \vee_{\mathcal{H}} k{:}s = k{:}(r \vee_k s)$	
$(\vee_{\mathcal{H}}2)$	$k_1{:}r \vee_{\mathcal{H}} k_2{:}s = \infty$	if $k_1 \neq k_2$, $k_1 \neq \mathsf{Nat}, k_2 \neq \mathsf{Nat}$
$(\vee_{\mathcal{H}}3)$	$k{:}r \vee_{\mathcal{H}} \mathsf{Nat}{:}n = k{:}(r \vee_k \iota_k(n))$	if $k \neq \mathsf{Nat}$
$(\vee_{\mathcal{H}}4)$	$\mathsf{Nat}{:}n \vee_{\mathcal{H}} k{:}r = k{:}(\iota_k(n) \vee_k r)$	if $k \neq \mathsf{Nat}$

where $\vee_k$ is the binary join of algebra $\mathcal{R}_k$. That is, the join of coeffects of the same kind is that in their coeffect algebra; the join of coeffects of different kinds is the trivial coeffect, apart from natural numers which can be mapped into coeffects of any kind.

Lemma 4.5. *For all $k_1{:}r, k_2{:}s, k_3{:}t \in |\mathcal{H}|$:*

1. *$k_1{:}r \preceq_{\mathcal{H}} k_1{:}r \vee_{\mathcal{H}} k_2{:}s$, $k_2{:}s \preceq_{\mathcal{H}} k_1{:}r \vee_{\mathcal{H}} k_2{:}s$*
2. *$k_1{:}r \preceq_{\mathcal{H}} k_3{:}t$, $k_2{:}s \preceq_{\mathcal{H}} k_3{:}t$ implies $k_1{:}r \vee_{\mathcal{H}} k_2{:}s \preceq_{\mathcal{H}} k_3{:}t$*

Sum and multiplication We define the sum operator $+_{\mathcal{H}}$ as follows:

$(+_{\mathcal{H}}1)$	$k{:}r +_{\mathcal{H}} k{:}s = k{:}(r +_k s)$	
$(+_{\mathcal{H}}2)$	$k_1{:}r +_{\mathcal{H}} k_2{:}s = \infty$	if $k_1 \neq k_2$, $k_1 \neq \mathsf{Nat}, k_2 \neq \mathsf{Nat}$
$(+_{\mathcal{H}}3)$	$k{:}r +_{\mathcal{H}} \mathsf{Nat}{:}n = k{:}(r +_k \iota_k(n))$	if $k \neq \mathsf{Nat}$
$(+_{\mathcal{H}}4)$	$\mathsf{Nat}{:}n +_{\mathcal{H}} k{:}r = k{:}(\iota_k(n) +_k r)$	if $k \neq \mathsf{Nat}$

where $+_k$ is the sum of the coeffect algebra $\mathcal{R}_k$. The definition is similar to that of the join operator.

Lemma 4.6. $\langle|\mathcal{H}|, \preceq_{\mathcal{H}}, +_{\mathcal{H}}, \mathsf{Nat}{:}0\rangle$ *is a partially ordered commutative monoid.*

We define the multiplication operator $\cdot_{\mathcal{H}}$ as follows:

$(\cdot_{\mathcal{H}}1)$	$k{:}r \cdot_{\mathcal{H}} k{:}s = k{:}(r \cdot_k t)$	if $k{:}r, k{:}s \neq \mathsf{Nat}{:}0$
$(\cdot_{\mathcal{H}}2)$	$k_1{:}r \cdot_{\mathcal{H}} k_2{:}s = \infty$	if $k_1 \neq k_2$, $k_1 \neq \mathsf{Nat}, k_2 \neq \mathsf{Nat}$
$(\cdot_{\mathcal{H}}3)$	$k{:}r \cdot_{\mathcal{H}} \mathsf{Nat}{:}n = k{:}(r \cdot_k \iota_k(n))$	if $n \neq \mathbf{0}$, $k \neq \mathsf{Nat}$
$(\cdot_{\mathcal{H}}4)$	$\mathsf{Nat}{:}n \cdot_{\mathcal{H}} k{:}r = k{:}(\iota_k(n) \cdot_k r)$	if $n \neq \mathbf{0}$, $k \neq \mathsf{Nat}$
$(\cdot_{\mathcal{H}}5)$	$\mathsf{Nat}{:}0 \cdot_{\mathcal{H}} k{:}r = k{:}r \cdot_{\mathcal{H}} \mathsf{Nat}{:}0 = \mathsf{Nat}{:}0$	

where $\cdot_k$ is the multiplication of the coeffect algebra $\mathcal{R}_k$. The definition is analogous to those of the join and sum operators, except that the result of multiplying by Nat:0 should be Nat:0, rather than being obtained mapping 0 in the kind k of the other argument, which would produce the **0** of that kind.

Lemma 4.7. $\langle|\mathcal{H}|, \preceq_{\mathcal{H}}, \cdot_{\mathcal{H}}, \mathsf{Nat}{:}1\rangle$ *is a partially ordered monoid.*

Theorem 4.8. $\mathcal{H} = \langle|\mathcal{H}|, \preceq_{\mathcal{H}}, +_{\mathcal{H}}, \cdot_{\mathcal{H}}, \mathsf{Nat}{:}0, \mathsf{Nat}{:}1\rangle$ *is a coeffect algebra.*

5. User-defined coeffects

We describe now an extension of the calculus supporting user-defined coeffects, reported in Fig. 3.

e	$::=$	$x \mid e.f \mid \texttt{new}\ C(es) \mid e.m(es) \mid C.m(es) \mid$ $\{T\ x = e;\ e'\} \mid \{T[\,\hat{v}\,]\ x = e;\ e'\} \mid \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2 \mid$ $e\ \texttt{instanceof}\ C \mid (C)e \mid \texttt{true} \mid \texttt{false} \mid \ldots$	expression
v	$::=$	$\texttt{new}\ C(vs) \mid x \mid \texttt{true} \mid \texttt{false}$	value

Fig. 3. Syntax with user-defined coeffects.

The only differences with the previous syntax are emphasized in grey: we include a non-annotated block, and in the annotated version the coeffect is in turn an expression of the calculus, notably a value, as detailed below.

We take a *stratified* approach, where the class table consists of two parts.

Standard class table The first part is a standard FJ class table, without coeffect annotations. Classes declared in this class table can be *coeffect classes*, that is, classes implementing methods corresponding to the ingredients of a coeffect algebra. In the calculus, we assume a predicate $\mathsf{coeff}(C)$ holding when C is a coeffect class. In the explicit syntax of the class table used to write examples, we will add a `coeffect` modifier before `class`. We assume that, if $\mathsf{coeff}(C)$ holds, then:

$\mathsf{mtype}(C, \texttt{leq}) = \star, C \to \texttt{boolean}$
$\mathsf{mtype}(C, \texttt{join}) = \star, C \to C$
$\mathsf{mtype}(C, \texttt{sum}) = \star, C \to C$
$\mathsf{mtype}(C, \texttt{mul}) = \star, C \to C$
$\mathsf{mtype}(C, \texttt{zero}) = \ \to C$
$\mathsf{mtype}(C, \texttt{one}) = \ \to C$

where, in the standard class table, we use $\star$ to denote that the method is an instance method.

We assume the following predefined coeffect classes:

```
abstract coeffect class Nat {
  Nat join(Nat x){if (this.leq(x)) x else this}
  static Nat zero(){new Zero()}
  static Nat one(){new Succ(Nat.zero())}
  }

class Zero extends Nat {
  boolean leq(Nat x){true}
  Nat sum(Nat x){x}
  Nat mult(Nat x){this}
}

class Succ extends Nat {
  Nat pred;
  boolean leq(Nat x){
    if (x instanceof Zero) false
    else pred.leq(((Succ) x).pred)
  }
  Nat sum(Nat x){new Succ(pred.sum(x))}
  Nat mult(Nat x){pred.mult(x).sum(x)}
}

coeffect class Triv {
   boolean leq(Triv t){true}
   Triv join(Triv t){this}
   Triv sum(Triv t){this}
   Triv mult(Triv t){this}
   static Triv zero(){new Triv()}
   static Triv one(){new Triv()}
}
```

Annotated class table The second part is a class table where coeffect annotations are (closed) values; we use the meta-variable $\hat{v}$ rather than v to suggest that they are expected to be *coeffect values*, that is, values of (a subclass of) a coeffect class. Coeffect annotations could be generalized to be arbitrary expressions; here we use this simpler assumption to make the presentation lighter. We will write $\vdash_{\mathsf{coeff}} \hat{v} : C$ to abbreviate $\emptyset \vdash \hat{v} : C$ and $\mathsf{coeff}(C)$, where these are judgments in the standard class table, and $\vdash_{\mathsf{coeff}} \hat{v}$ if $\vdash_{\mathsf{coeff}} \hat{v} : C$ for some C, that is, $\hat{v}$ is a coeffect value.

In this class table, we have that the enriched method type, returned by function mtype, is of shape:

- either $\hat{v}_0, T_1^{\hat{v}_1} \ldots T_n^{\hat{v}_n} \rightarrow T$
- or $T_1^{\hat{v}_1} \ldots T_n^{\hat{v}_n} \rightarrow T$, meaning that the method is static.

The class table is stratified in the sense that the second part can use classes declared in the first part (the standard class table), but not conversely. Notably, as said above, coeffect annotations in the second class table are values typechecked in the standard part; moreover, standard classes can be used in the annotated class table assuming everywhere an implicit trivial annotation, that is, `new Triv()`.

For a given class table, the parametric type system defined in Fig. 2 is instantiated taking the coeffect algebra of heterogeneous coeffects obtained with the construction in Section 4, starting from the following family of coeffect algebras:

- the kinds are the names of declared coeffect classes (including the predefined `Nat` and `Triv`)
- for each kind (coeffect class), the elements of the carrier are the corresponding coeffect values, and the partial order and the operations are derived from methods, as will be detailed in the following.

Note that, since overloading is prevented by assumption (T-INH-METH), a coeffect class cannot be extended by another coeffect class. Hence, the coeffect class of each coeffect value is uniquely determined.

Also note that a coeffect value $\hat{v}$ such that $\vdash_{\mathsf{coeff}} \hat{v} : C$ corresponds to a kinded coeffect $C{:}\hat{v}$ as abstractly defined in Section 4. Accordingly with this remark, we will abbreviate `new Zero()` and `new Succ(new Zero())` by `Nat`:0 and `Nat`:1, respectively.

The typing rules obtained by this instantiation are all reported, for reader's convenience, in Fig. 4. Points where it is made explicit that coeffects are values of the calculus are emphasized in grey. In particular, note that the one coeffect of the heterogeneous coeffect algebra is `Nat`:1, and that, in rule (T-BLOCK), it must be checked that the annotation is actually a coeffect value.

Provided that code defining coeffects is terminating (see below), the typing rules directly lead to a typechecking algorithm. Indeed:

- the type of an expression, if any, can be computed in the standard way, notably subsumption can be replaced by explicit subtyping conditions for arguments in rules (T-NEW), (T-INVK), and (T-ST-INVK), and initialization expression in rule (T-BLOCK), and for arguments/result in conditions (T-METH) and (T-ST-METH)
- the coeffects can be computed bottom-up, starting from the rules for variable and constants, also thanks to the fact that, when an upper bound of coeffects is required, as in rule (T-IF) and side conditions of rules (T-INVK), (T-ST-

$$\text{(T-SUB)}\ \frac{\Gamma \vdash e : T \quad \Gamma \preceq \Gamma'}{\Gamma' \vdash e : T'}\ T \leq T' \qquad \text{(T-VAR)}\ \frac{}{x :_{\mathtt{Nat{:}1}} T \vdash x : T}$$

$$\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash e : C}{\Gamma \vdash e.f_i : C_i}\ \begin{array}{l}\mathsf{fields}(C) = C_1\ f_1; \ldots C_n\ f_n;\\ i \in 1..n\end{array}$$

$$\text{(T-NEW)}\ \frac{\Gamma_i \vdash e_i : C_i \quad \forall i \in 1..n}{\Gamma_1 + \ldots + \Gamma_n \vdash \mathtt{new}\ C(e_1, \ldots, e_n) : C}\ \mathsf{fields}(C) = C_1\ f_1; \ldots C_n\ f_n;$$

$$\text{(T-INVK)}\ \frac{\Gamma_i \vdash e_i : C_i \quad \forall i \in 0..n}{\hat{v}'_0 \cdot \Gamma_0 + \ldots + \hat{v}'_n \cdot \Gamma_n \vdash e_0.m(e_1, \ldots, e_n) : T}\ \begin{array}{l}\mathsf{mtype}(C_0, m) = \hat{v}_0, T_1^{\hat{v}_1} \ldots T_n^{\hat{v}_n} \to T\\ \hat{v}'_i = \hat{v}_i \vee \mathtt{Nat{:}1} \quad \forall i \in 0..n\end{array}$$

$$\text{(T-ST-INVK)}\ \frac{\Gamma_i \vdash e_i : C_i \quad \forall i \in 1..n}{\hat{v}'_1 \cdot \Gamma_1 + \ldots + \hat{v}'_n \cdot \Gamma_n \vdash C.m(e_1, \ldots, e_n) : T}\ \begin{array}{l}\mathsf{mtype}(C_0, m) = T_1^{\hat{v}_1} \ldots T_n^{\hat{v}_n} \to T\\ \hat{v}'_i = \hat{v}_i \vee \mathtt{Nat{:}1} \quad \forall i \in 1..n\end{array}$$

$$\text{(T-BLOCK)}\ \frac{\begin{array}{l}\vdash_{\mathsf{coeff}} \hat{v}\\ \Gamma \vdash e : T \qquad \Gamma', x :_{\hat{v}} T \vdash e' : T'\end{array}}{\hat{v}' \cdot \Gamma + \Gamma' \vdash \{T[\hat{v}]\ x = e;\ e'\} : T'}\ \hat{v}' = \hat{v} \vee \mathtt{Nat{:}1}$$

$$\text{(T-IF)}\ \frac{\Gamma \vdash e : \mathtt{boolean} \quad \Delta_1 \vdash e_1 : T \quad \Delta_2 \vdash e_2 : T}{\Gamma + (\Delta_1 \vee \Delta_2) \vdash \mathtt{if}\,(e)\,e_1\,\mathtt{else}\ e_2 : T}$$

$$\text{(T-INSTOF)}\ \frac{\Gamma \vdash e : D}{\Gamma \vdash e\ \mathtt{instanceof}\ C : \mathtt{boolean}} \qquad \text{(T-CAST)}\ \frac{\Gamma \vdash e : D}{\Gamma \vdash (C)e : C}\ C \leq D$$

$$\text{(T-TRUE)}\ \frac{}{\vdash \mathtt{true} : \mathtt{boolean}} \qquad \text{(T-FALSE)}\ \frac{}{\vdash \mathtt{false} : \mathtt{boolean}}$$

Fig. 4. Type-and-coeffect system with user-defined coeffects.

INVK), and (T-BLOCK), it is computed by using the join operator; subsumption can be analogously replaced by explicit subtyping conditions on coeffect contexts for initialization expression in rule (T-BLOCK), and for arguments/result in conditions (T-METH) and (T-ST-METH).

Example 5.1 *(Affinity).* Affinity coeffects could be implemented as follows:

```
abstract coeffect class Affinity {
  Affinity join(Affinity x){if (this.leq(x)) x else this}
  static Affinity zero(){new ZeroA()}
  static Affinity one(){new One()}
  }

class ZeroA extends Affinity {
  boolean leq(Affinity x){true}
  Affinity sum(Affinity x){x}
  Affinity mult(Affinity x){this}
}
class One extends Affinity {
  boolean leq(Affinity x){!(x instanceof ZeroA)}
  Affinity sum(Affinity x){
    if (x instanceof ZeroA) this else new Omega()
  }
  Affinity mult(Affinity x){x}
}
class Omega extends Affinity {
  boolean leq(Affinity x){x instanceof Omega}
  Affinity sum(Affinity x){this}
  Affinity mult(Affinity x){
    if (x instanceof ZeroA) x else this
  }
}
```

and the previous Example 3.1 becomes as follows:

```
class Pair {A fst; A snd;}
class A {
  A drop [new ZeroA()] () {new A()}
  A identity [new One()] () {this}
  Pair duplicate [new Omega()] () { new Pair(this,this)}
}
```

Example 5.2 *(Privacy levels).* The following coeffect class `Privacy` provides a way to specify the privacy level of data. In this case, the coeffects form a three point lattice: `Public`, `Private` and `Irrelevant` with `zero` being `Irrelevant`, `one` being `Private` and order `Irrelevant`$\preceq$`Private`$\preceq$`Public`. Sum is the join and multiplication is defined by $r_1 \cdot r_2 =$`Irrelevant` if either $r_1 =$`Irrelevant` or $r_2 =$`Irrelevant`, otherwise $r_1 \cdot r_2 = r_1 \vee r_2$.

```
abstract coeffect class Privacy {
  Privacy join(Privacy x){if (this.leq(x)) x else this}
  Privacy sum(Privacy x){this.join(x)}
  static Privacy zero(){new Irrelevant()}
  static Privacy one(){new Private()}
  }

class Irrelevant extends Privacy {
  boolean leq(Privacy x){true}
  Privacy mult(Privacy x){this}
}
class Private extends Privacy {
  boolean leq(Privacy x){!(x instanceof Irrelevant)}
  Privacy mult(Privacy x){this.join(x)}
  }

}
class Public extends Privacy {
  boolean leq(Privacy x){x instanceof Public}
  Privacy mult(Privacy x){this.join(x)}
}
```

Example 5.3 *(Pairs).* The following example shows that the programmer can also define coeffect classes constructed by combining other coeffect classes. The class `APPair` implements coeffects which are pairs of affinity coeffects and privacy levels.

```
coeffect class APPair {Affinity left; Privacy right;
  boolean leq(APPair p){
    this.left.leq(p.left)&&this.right(p.right)
  }
  APPair join(APPair p){
    new APPair(this.left.join(p.left),this.right.join(p.right))
  }
  APPair sum(APPair p){
    new APPair(this.left.sum(p.left),this.right.sum(p.right))
  }
  APPair mult(APPair p){
    new APPair(this.left.mul(p.left),this.right.mul(p.right))
  }
  static APPair zero(){
    new APPair(Affinity.zero(),Privacy.zero()
  }
  static APPair one(){
    new APPair(Affinity.one(),Privacy.one())
  }
}
```

In full Java, where a coeffect class could be expressed as a class implementing a certain generic interface, as described later, we could even define a generic class implementing pairs of arbitrary coeffects.

Following the stratified approach, we expect typechecking to be performed in two steps:

1. The standard class table, containing declarations of coeffect classes, is typechecked by the standard compiler.

2. Code containing coeffect annotations written in Java is typechecked accordingly to the type-and-coeffect system in Fig. 4, where the underlying coeffect algebra is obtained by composing, with the construction described in Section 4, the user-defined coeffect algebras, whose operations are computed by executing user-defined methods in such class, as detailed below.

Recall that, with the usual notations and terminology of reduction relations, $\longrightarrow^\star$ denotes the transitive and reflexive closure of $\longrightarrow^\star$, and e' is a *normal form* of e if $e \longrightarrow^\star e'$ and there is no e'' such that $e' \longrightarrow e''$. It is easy to see that the FJ reduction relation is deterministic, hence the normal form of e, if any, is unique. However, there can be no normal form at all, since the reduction of e could be non-terminating. We assume that methods `leq`, `join`, `sum`, `mult`, `zero`, and `one` in coeffect classes always terminate, so that the notation $\mathsf{nf}(e)$ for the normal form of e in the definitions below is well-defined. Then, operations on coeffects of kind C, that is, coeffect values of class C, are defined as follows:

Leq	$\hat{v}_1 \preceq_C \hat{v}_2 = \mathsf{nf}(\hat{v}_1.\mathtt{leq}(\hat{v}_2))$
Join	$\hat{v}_1 \vee_C \hat{v}_2 = \mathsf{nf}(\hat{v}_1.\mathtt{join}(\hat{v}_2))$
Sum	$\hat{v}_1 +_C \hat{v}_2 = \mathsf{nf}(\hat{v}_1.\mathtt{sum}(\hat{v}_2))$
Multiplication	$\hat{v}_1 \cdot_C \hat{v}_2 = \mathsf{nf}(\hat{v}_1.\mathtt{mul}(\hat{v}_2))$
Zero	$\mathbf{0}_C = \mathsf{nf}(C.\mathtt{zero}())$
One	$\mathbf{1}_C = \mathsf{nf}(C.\mathrm{one}())$

Note that the unique homomorphism ι_C from the initial coeffect algebra to the coeffect algebra implemented by C turns out to be computed using the `zero`, `one`, and `sum` methods, as follows, where $\iota_C(n)$ is the coeffect of class C corresponding to n:

$$\iota_C(\mathtt{Nat{:}0}) = \mathsf{nf}(C.\mathtt{zero}()) \qquad \iota_C(\mathtt{Nat{:}1}) = \mathsf{nf}(C.\mathrm{one}())$$
$$\iota_C(\mathtt{new\ Succ}(n)) = \mathsf{nf}(\iota_C(n).\mathrm{sum}(C.\mathrm{one}()))$$

For the whole process to work correctly, the following are responsabilities of the programmer:

- Code defining coeffects should be *terminating*, since, as described above, the second typechecking step requires to *execute* code typechecked in the first step.
- Coeffect classes should satisfy the required axioms, e.g., the sum derived from `sum` methods should be commutative and associative. The same happens, for instance, in Haskell, when one defines instances of `Functor` or `Monad`.

Implementations could use in a parametric way auxiliary tools, notably a termination checker to prevent divergence in methods implementing grade operations, and/or a verifer to ensure that they provide the required properties.

We end this section outlining how the approach could be implemented in full Java. We omit access modifiers to make the code lighter.

In the calculus, we abstractly modeled coeffect classes as classes required to implement certain methods. In the full Java language, such requirement could be imposed by defining the following generic interfaces:

```
interface Coeffect<T extends Coeffect<T>> {
  boolean leq(T x);
  T join(T x);
  T sum(T x);
  T mult(T x);
}

interface CoeffectFactory<T extends Coeffect<T>>{
  T zero();
  T one();
}
```

For instance, the implementation of affinity coeffects would become as follows, with subclasses as before:

```
abstract class Affinity implements Coeffect<Affinity>{
  Affinity join(Affinity x){
    if (this.leq(x)) x else this
  }
}

class AffinityFactory implements CoeffectFactory<Affinity>{
  Affinity zero(){new ZeroA()}
  Affinity one(){new One()}
}
```

```
class A {}
class Triple {
  Msg msg; OptData data;OutPrivChannel cont;
}
class Unit {}
class OptData {}
class Some extends OptData {
  A f;
}
class None extends OptData {}
class Msg {}
class NextData extends Msg {}
class Stop extends Msg {}
class OK extends Msg {}
class KO extends Msg {}
```

Fig. 5. Some auxiliary classes.

```
class OutPrivChannel{
  Unit send [new One()](Msg msg,OptData [new Private()] data,
    OutPrivChannel [new One()] cont) {...}
}
class InPrivChannel{
  Triple rcv [new One()] () {...}
}
```

Fig. 6. Channels.

Note that the implementation, as expected, depends on the features of the target language; for instance, in Scala we would likely use case classes, and, turning to a different paradigm, in Haskell we could express `Coeffect` as a typeclass, and coeffect algebras as its instances.

6. A programming example

We show a more significant programming example, which illustrates how different kinds of coeffects can be helpful in the same program; indeed it uses the coeffect classes `Affinity` and `Privacy` defined before. In this code we omit curly brackets when the body of a block is a block in turn, and we use sequences, which can be encoded as blocks where the local variable is unused.

The example illustrates a client-server application in which a client sends some data to a server using a session-based approach. We take inspiration from the encoding of sessions into the π-calculus with variants and linear I/O types of [11]. In our framework, where the zero coeffect is the least element, linear types are approximated as affine types. We assume to have some classes implementing the data and messages exchanged, see Fig. 5.

In Fig. 6 are the classes implementing affine input and output channels over which we can send a message and some private data. The affinity of the input and output channels is expressed by annotating the receiver of the send and receive methods with `new One()` of the `Affinity` coeffects. The `send` method of the class `OutPrivChannel` takes as input, in addition to the message and the data to be sent, an output channel that will be used, by whoever is receiving the message, to continue the interaction, that is, to send back a message. On the channels only private data can be sent. This is enforced by the annotation `new Private()` of the parameter `data`. The `new One()` annotation of the parameter `cont` asserts that the argument must be an affine channel. The method `rcv` of the class `InPrivChannel` returns a triple containing a message, a data and an output channel that will be used, by whoever receives the message, to continue the conversation.

The class `Server` of Fig. 7 implements a server which waits on a channel for a triple whose message should be either `NextData` or `Stop`.
If the server receives `NextData`, then, after creating a pair of input and output channels, it sends to whoever sent the triple (by using the output channel received) a triple containing the message `new OK()`, no data and the output channel created. Then, after processing the received data, the server continues the interaction by waiting on the input channel just created, which is paired with the output channel sent. This is done by the recursive call `main(inCh)`.
If the server receives `Stop`, then it stops returning `new OK()` (we use this message also to signal that the protocol ended successfully).
If the server receives any other message, then it stops returning `new KO()`, meaning failure of the exchange. Note that the server receives the initial message from the client on the channel which is the parameter of the method. After receiving the message, the channel cannot be used any longer.

```
class Server {
  Msg main(InPrivChannel [new One()] ch) {
    Triple [new Omega()] res = ch.rcv();
    Msg [new Omega()] msg = res.msg;
    if (msg instanceof NextData) {
      OutPrivChannel [new One()] ch1 = res.cont;
      OutPrivChannel [new One()] outCh = new OutPrivChannel();
      InPrivChannel [new One()] inCh = new InPrivChannel();
      ch1.send(new OK(), new None(), outCh);
      // process res.data
      main(inCh)
    }
    else if (msg instanceof Stop) new OK() else new KO()
  }
}
```

Fig. 7. The server.

```
class Client {
  Msg main(OutPrivChannel [new One()] ch) {
    if (/*Client decides to send data*/) {
      OutPrivChannel [new One()] outCh = new OutPrivChannel();
      InPrivChannel [new One()] inCh = new InPrivChannel();
      ch.send(new NextData(), new Some(new A()), outCh);
      Triple [new Omega()] res = inCh.rcv();
      Msg [new Omega()] msg = res.msg;
      OutPrivChannel [new One()] ch1 = res.cont;
      if (msg instanceof OK) main(ch1) else new KO()
    }
    else { // Client decides to stop
      ch.send(new Stop(), new None(), null);
      new OK()
    }
  }
}
```

Fig. 8. The client.

```
OutPrivChannel [new One()] outCh = new OutPrivChannel();
InPrivChannel [new One()] inCh = new InPrivChannel();
  new Client().main(outCh) | new Server().main(inCh)
```

Fig. 9. Starting the protocol.

The class `Client` of Fig. 8 implements a client of the previous server.
If the client wants to send more data to the server, after creating a new pair of input and output channels, it sends, on the channel it was given, a triple consisting of the message `NextData`, a new data and the created output channel. Then the client waits to receive a message on the input channel paired with the one sent. If the received message is `OK`, meaning that the server correctly processed the sent data, then the client starts again using the channel received from the server. This is done by the recursive call `main(ch1)`. If the received message is `KO`, meaning that the server could not process the sent data, then the client stops returning `new KO()`.
If the client does not want to send more data to the server, then it stops returning `new OK()`.

The computation of the server and the client is started, see Fig. 9, by creating a pair of input and output channels and sending the output channel to the client and the input channel to the server. The fact that the channels are affine ensures that they will be used to realise the wanted binary session. Here we assume to have a parallel composition operator.

7. Related work

Our work has been inspired by Granule [19], a functional language equipped with graded modal types, where different kinds of coeffects (grades) can be used at the same time, including naturals for exact usage, security levels, intervals, infinity, and products of coeffects.

We owe to Granule the overall objective of exploiting coeffects in a programming language, pursued here in a different paradigm, and the idea of allowing different kinds of coeffects to coexist. Concerning this latter objective, in this paper we push forward the Granule approach, since we do not want the available coeffects to be fixed, but definable by the programmer. To this aim we define the coeffect algebra of heterogeneous coeffects in Section 4. The solution offered by this construction is rudimentary, in the sense that combination of coeffects of different kinds always leads to the trivial coeffect, apart from natural numbers which can be properly combined with others through their embedding. On the contrary, by relying on the fact that the available coeffects are known in advance, Granule can provide ad-hoc combinations. However, our approach has the important advantage to be *modular*, in the sense that combination of several coeffect algebras is shown to produce a coeffect algebra, allowing us to reuse the general meta-theory, e.g., to prove soundness, rather than providing an ad-hoc proof. The simple construction of this paper is a first step towards more flexible definitions, as discussed in the next section.

Other practical programming languages incorporating (a variant of) coeffects are Idris 2 [7] and Linear Haskell [3]. The first is a dependently typed functional language implementing an instance of quantitative type theory [2], thus serving also as a proof assistant. Differently from Granule and this paper, Idris 2 uses just a single semiring of coeffects consisting of 0, 1 and ω, as the main goal is to identify code not needed at runtime. The second adds to Haskell first-order linearly typed functions and data structures. Function types are annotated with the multiplicity (a natural number) of the argument that they require to produce their output. In our setting this would be using as coeffects the semiring of natural numbers. We conjecture that, with a construction similar to the one we propose in this paper, Linear Haskell could support user-defined coeffects. As mentioned before, they would be, rather than values of a (subclass of) a coeffect class as in our calculus, values of instances of a predefined `Coeffect` typeclass offering the ingredients of coeffect algebras.

Turning now to the literature on coeffects in general, the notion was firstly introduced by [20] and further analyzed by [21]. In particular, [21] 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 [8], who developed a framework for structural coeffects for a functional language. Many advances have then been made to combine coeffects with other programming features, such as computational effects [12,19,10], dependent types [2,9,18], and polymorphism [1]. Other graded type systems are explored in [2,13,1], also combining effects and coeffects [12,19]. In all these papers, the process of tracking usage through grades is a powerful method of instrumenting type systems with analyses of irrelevance and linearity that have practical benefits like erasure of irrelevant terms (resulting in speed-up) and compiler optimizations (such as in-place update).

In [18] and [22] it was observed that contexts in a structural coeffect system form a module over the semiring of grades, even though they do not use this structure in its full generality, restricting themselves to free modules, that is, to structural coeffect systems. Recently, [6] shows a significant non-structural instance, namely, a coeffect system to track sharing in the imperative paradigm.

8. Conclusion

We proposed a Java-like calculus supporting, in variable declarations, coeffect annotations, allowing to express how variables should be used. We formally defined the type system and proved subject reduction, which includes preservation of coeffects, and provided several examples. Moreover, we have shown that coeffects can be *heterogeneous*, in the sense that different kinds of coeffects can be used in the same program, and they do not need to be fixed once and for all. Indeed, we provided a formal construction leading to a unique coeffect algebra, where, roughly, combining coeffects of different kinds gives the trivial coeffect. Finally, we proposed an extension of the calculus where programmers are able to define their own classes implementing coeffect algebras, so that coeffect annotations are themselves expressions of the calculus, similarly to what happens with user-defined Java exceptions.

In further work [5], we investigated three further developments of the contribution presented in this paper. First of all, the coeffect algebra of heterogeneous coeffects defined in Section 4 is constructed taking the simplest choice, corresponding to assume that the programmer "does not know" how to combine coeffects of different kinds. In [5], we designed a more general framework where, depending on some additional parameters, a coeffect algebra of heterogeneous coeffects can be constructed in many ways. Then, a limitation of the proposal in this paper is that, whereas it is possible to specify how a variable should be used (e.g., a parameter should be used at most once in a method's body), it is not possible to do the same for *the result of an expression* (e.g., the result of a method). The variant of the calculus in [5], equipped with *graded modal types*, which are types annotated with coeffects (grades) [8,19,10], also similar to types annotated with *modifiers* or *capabilities* [16,15,14], overcomes this limitation. Finally, the soundness theorem proved in this paper states that coeffects are preserved, but does *not* express the fact that coeffects actually overapproximate the usage of resources, since the latter is not modeled in the standard reduction semantics. To this end, we developed in [5] an *instrumented* semantics keeping track of resource consumption, as done in [9].

Coeffects considered in this paper are *structural*, in the sense that they are expressed and computed on a per-variable basis. However, in some cases the coeffect, expressing how an expression uses external resources, cannot be captured by just assigning independent *scalar* coeffects to single variables, but should be assigned to the whole context [21]. In our work, this would correspond to allow a "global" annotation in a method's signature.

Moreover, expressive power could be added by allowing variables in coeffect annotations, so to specify, e.g., that a variable should be used no more than a certain number computed at runtime. This approach would require first the study of *dependent* coeffects on the foundational side, which, to the best of our knowledge, has not been done yet.

On the more applicative side, we could investigate how the proposal scales to realistic subsets of Java, and possible implementations. As mentioned, an interesting point is that implementations could use in a parametric way auxiliary tools. The application of the approach to different paradigms, e.g., in Haskell as sketched before, is also an interesting direction.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X) and has the financial support of the University of Eastern Piedmont.

Appendix A. Proofs of Section 4

Proof of Lemma 4.4. We have to prove that $\preceq_{\mathcal{H}}$ is reflexive, transitive and antisymmetric.

Reflexivity. Let $k{:}r \in |\mathcal{H}|$. Since $\preceq_k$ is reflexive, we have $r \preceq_k r$. By ($\preceq_{\mathcal{H}}$ 1) we get $k{:}r \preceq_{\mathcal{H}} k{:}s$.

Transitivity. Let $k_1{:}r, k_2{:}s, k_3{:}t \in |\mathcal{H}|$ and $k_1{:}r \preceq_{\mathcal{H}} k_2{:}s$ and $k_2{:}s \preceq_{\mathcal{H}} k_3{:}t$. We split cases on the definition of $k_1{:}r \preceq_{\mathcal{H}} k_2{:}s$.

($\preceq_{\mathcal{H}}$ 1) $k_1 = k_2$ and $k_2 \neq \mathsf{Triv}$
If $k_2 = k_3$ then, by transitivity of $\preceq_{k_1}$ and ($\preceq_{\mathcal{H}}$ 1), we can conclude $k_1{:}r \preceq_{\mathcal{H}} k_3{:}t$. If $k_3 = \mathsf{Triv}$ then $k_1{:}r \preceq_{\mathcal{H}} k_3{:}t$ holds by ($\preceq_{\mathcal{H}}$ 2). If $k_2 = \mathsf{Nat}$ and $k_3 \neq \mathsf{Nat}$, then we have $\iota_{k_3}(s) \preceq_{k_3} t$. Since ι_{k_3} is an homomorphism, $r \preceq_{\mathsf{Nat}} s$ implies $\iota_{k_3}(r) \preceq_{k_3} \iota_{k_3}(s)$, so by transitivity of $\preceq_{k_3}$ we have $\iota_{k_3}(r) \preceq_{k_3} t$. By ($\preceq_{\mathcal{H}}$ 3), we can conclude $k_1{:}r \preceq_{\mathcal{H}} k_3{:}t$.

($\preceq_{\mathcal{H}}$ 2) $k_2 = \mathsf{Triv}$
In this case we know that $k_3 = \mathsf{Triv}$, so $k_1{:}r \preceq_{\mathcal{H}} k_3{:}t$ by ($\preceq_{\mathcal{H}}$ 2).

($\preceq_{\mathcal{H}}$ 3) $k_1 = \mathsf{Nat}$ and $k_2 \neq \mathsf{Nat}$
We have $\iota_{k_2}(r) \preceq_{k_2} s$. If $k_2 = k_3$ then, by transitivity of $\preceq_{k_2}$, we have $\iota_{k_2}(r) \preceq_{k_2} t$ and, by ($\preceq_{\mathcal{H}}$ 3), $k_1{:}r \preceq_{\mathcal{H}} k_3{:}t$. If $k_3 = \mathsf{Triv}$ then we have the thesis by ($\preceq_{\mathcal{H}}$ 2).

Antisymmetry. Let $k_1{:}r, k_2{:}s \in |\mathcal{H}|$ and $k_1{:}r \preceq_{\mathcal{H}} k_2{:}s$ and $k_2{:}s \preceq_{\mathcal{H}} k_1{:}r$. Then it must be $k_1 = k_2$. Therefore the thesis follows by the antisymmetry of $\preceq_{k_1}$ and ($\preceq_{\mathcal{H}}$ 1). □

Proof of Lemma 4.5. Proof of Item 1. We split cases on the definition of $k_1{:}r \vee_{\mathcal{H}} k_2{:}s$.

($\vee_{\mathcal{H}}$1) $k_1 = k_2$
The thesis follows since $\preceq_{k_1}$ has this property.

($\vee_{\mathcal{H}}$2) $k_1 \neq k_2$, $k_1 \neq \mathsf{Nat}$, $k_2 \neq \mathsf{Nat}$
We have $k_1{:}r \vee_{\mathcal{H}} k_2{:}s = \mathsf{Triv}{:}\infty$, so, by ($\preceq_{\mathcal{H}}$ 2), we have the thesis.

($\vee_{\mathcal{H}}$3) $k_1 \neq \mathsf{Nat}$, $k_2 = \mathsf{Nat}$
The thesis follows by ($\vee_{\mathcal{H}}$3), ($\preceq_{\mathcal{H}}$ 3) and since $\preceq_{k_1}$ has this property.

($\vee_{\mathcal{H}}$4) $k_1 \neq \mathsf{Nat}$, $k_2 = \mathsf{Nat}$
The proof is similar the one above.

Proof of Item 2. We split cases on the definition of $k_1{:}r \preceq_{\mathcal{H}} k_3{:}t$.

($\preceq_{\mathcal{H}}$ 1) $k_1 = k_3$ and $k_1 \neq \mathsf{Triv}$
If $k_1 = k_2$ then the thesis follows since $\preceq_{k_1}$ has this property. If $k_2 = \mathsf{Nat}$ then we know $k_1{:}r \vee_{\mathcal{H}} k_2{:}s = k_1{:}(r \vee_{k_1} \iota_{k_1}(s))$. By ($\preceq_{\mathcal{H}}$ 1) we have $r \preceq_{k_1} t$ and by ($\preceq_{\mathcal{H}}$ 3) we have $\iota_{k_1}(s) \preceq_{k_1} t$. By these considerations we know $r \vee_{k_1} \iota_{k_1}(s) \preceq_{k_1} t$ and so by ($\preceq_{\mathcal{H}}$ 1) we have the thesis.

($\preceq_{\mathcal{H}}$ 2) $k_3 = \mathsf{Triv}{:}\infty$
The thesis follows from ($\preceq_{\mathcal{H}}$ 2).

($\preceq_{\mathcal{H}}$ 3) $k_1 = \mathsf{Nat}$ and $k_3 \neq \mathsf{Nat}, \mathsf{Triv}$
If $k_2 = \mathsf{Nat}$ we have $k_1{:}r \vee_{\mathcal{H}} k_2{:}s = \mathsf{Nat}{:}(r \vee_{\mathsf{Nat}} s)$. By ($\preceq_{\mathcal{H}}$ 3) we have $\iota_{k_3}(r) \preceq_{k_3} t$ and $\iota_{k_3}(s) \preceq_{k_3} t$, so we know $\iota_{k_3}(r) \vee_{k_3} \iota_{k_3}(s) \preceq_{k_3} t$. Since ι is an homomorphism, $\iota_{k_3}(r) \vee_{k_3} \iota_{k_3}(s) = \iota_{k_3}(r \vee_{\mathsf{Nat}} s)$, so by ($\preceq_{\mathcal{H}}$ 3) and ($\vee_{\mathcal{H}}1$) we have the thesis. The proof for the case $k_2 = k_3$ is analogous to the proof for the case $k_1 = k_3$, $k_1 \neq \mathsf{Triv}$ and $k_2 = \mathsf{Nat}$. □

Proof of Lemma 4.6. We have to prove that $+_{\mathcal{H}}$ is commutative, associative and monotonic with respect to $\preceq_{\mathcal{H}}$ and that $\mathsf{Nat}{:}0$ is the identity of $+_{\mathcal{H}}$. In particular, given $k_1{:}r$, $k_2{:}s$, $k_3{:}t$:

Commutativity. $k_1{:}r +_{\mathcal{H}} k_2{:}s = k_2{:}s +_{\mathcal{H}} k_1{:}r$. We split cases on the definition of $k_1{:}r +_{\mathcal{H}} k_2{:}s$.

($+_{\mathcal{H}}1$) $k_1 = k_2$
The thesis follows since $+_{k_1}$ is commutative.

($+_{\mathcal{H}}2$) $k_1 \neq k_2, k_1 \neq \mathsf{Nat}.k_2 \neq \mathsf{Nat}$
The thesis follows since $k_1{:}r +_{\mathcal{H}} k_2{:}s = \infty$ and $k_2{:}s +_{\mathcal{H}} k_1{:}r = \infty$.

($+_{\mathcal{H}}3$) $k_1 = \mathsf{Nat}, r = n, k_2 \neq \mathsf{Nat}$
We have $k_1{:}r +_{\mathcal{H}} k_2{:}s = \mathsf{Nat}{:}n +_{\mathcal{H}} k_2{:}s = k_2{:}(\iota_{k_2}(n) +_{k_2} s)$. By commutative property of $+_{k_2}$ we have $k_2{:}(\iota_{k_2}(n) +_{k_2} s) = k_2{:}(s +_{k_2} \iota_{k_2}(n)) = k_2{:}s +_{\mathcal{H}} \mathsf{Nat}{:}n$.

($+_{\mathcal{H}}4$) $k_2 = \mathsf{Nat}, s = n, k_1 \neq \mathsf{Nat}$
Analogous to the case above.

Associativity. $(k_1{:}r +_{\mathcal{H}} k_2{:}s) +_{\mathcal{H}} k_3{:}t = k_1{:}r +_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t)$. We split cases on the definition of $k_1{:}r +_{\mathcal{H}} k_2{:}s$.

($+_{\mathcal{H}}1$) $k_1 = k_2$
If $k_2 = k_3$ the thesis follows from the associativity of $+_{k_2}$. If $k_3 \neq k_2, k_3 \neq \mathsf{Nat}, k_2 \neq \mathsf{Nat}$ then we have the thesis since $(k_1{:}r +_{\mathcal{H}} k_2{:}s) +_{\mathcal{H}} k_3{:}t = k_1{:}(r +_{k_1} s) +_{\mathcal{H}} k_3{:}t = \mathsf{Triv}{:}\infty$ and $k_1{:}r +_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = k_1{:}r +_{\mathcal{H}} \mathsf{Triv}{:}\infty = \mathsf{Triv}{:}\infty$. If $k_1, k_2 = \mathsf{Nat}$ and $k_3 \neq \mathsf{Nat}$ we have $(k_1{:}r +_{\mathcal{H}} k_2{:}s) +_{\mathcal{H}} k_3{:}t = \mathsf{Nat}{:}(r +_{\mathsf{Nat}} s) +_{\mathcal{H}} k_3{:}t = k_3{:}(\iota_{k_3}(r +_{\mathsf{Nat}} s) +_{k_3} t)$. Since ι is an homomorphism, $k_3{:}(\iota_{k_3}(r +_{\mathsf{Nat}} s) +_{k_3} t) = k_3{:}((\iota_{k_3}(r) +_{k_3} \iota_{k_3}(s)) +_{k_3} t)$ and by the associativity of k_3 we have $k_3{:}((\iota_{k_3}(r) +_{k_3} \iota_{k_3}(s)) +_{k_3} t) = k_3{:}(\iota_{k_3}(r) +_{k_3} (\iota_{k_3}(s) +_{k_3} t)) = \mathsf{Nat}{:}r +_{\mathcal{H}} k_3{:}(\iota_{k_3}(s) +_{k_3} t) = \mathsf{Nat}{:}r +_{\mathcal{H}} (\mathsf{Nat}{:}s +_{\mathcal{H}} k_3{:}t)$. If $k_3 = \mathsf{Nat}$ and $k_1 \neq \mathsf{Nat}$ the proof is analogous.

($+_{\mathcal{H}}2$) $k_1 \neq k_2$ and $k_1, k_2 \neq \mathsf{Nat}$
If $k_3 \neq k_1$, $k_3 \neq k_2$ and $k_3 \neq \mathsf{Nat}$, then the thesis follows since $(k_1{:}r +_{\mathcal{H}} k_2{:}s) +_{\mathcal{H}} k_3{:}t = k_1{:}r +_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = \mathsf{Triv}{:}\infty$. If $k_3 = k_1$ then $(k_1{:}r +_{\mathcal{H}} k_2{:}s) +_{\mathcal{H}} k_3{:}t = \mathsf{Triv}{:}\infty +_{\mathcal{H}} k_3{:}t = \mathsf{Triv}{:}\infty = k_1{:}r +_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = k_1{:}r +_{\mathcal{H}} \mathsf{Triv}{:}\infty$. If $k_2 = k_3$ then $(k_1{:}r +_{\mathcal{H}} k_2{:}s) +_{\mathcal{H}} k_3{:}t = \mathsf{Triv}{:}\infty +_{\mathcal{H}} k_3{:}t = k_1{:}r +_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = k_1{:}r +_{\mathcal{H}} k_2{:}(s +_{k_2} t) = \mathsf{Triv}{:}\infty$. If $k_3 = \mathsf{Nat}$ then $(k_1{:}r +_{\mathcal{H}} k_2{:}s) +_{\mathcal{H}} k_3{:}t = \mathsf{Triv}{:}\infty +_{\mathcal{H}} k_3{:}t = \mathsf{Triv}{:}(\mathsf{Triv}{:}\infty +_{\mathsf{Triv}} \iota_{\mathsf{Triv}}(t)) = \mathsf{Triv}{:}\infty$.

($+_{\mathcal{H}}3$) $k_1 = \mathsf{Nat}, k_2 \neq \mathsf{Nat}$
If $k_3 = k_2$ then $(k_1{:}r +_{\mathcal{H}} k_2{:}s) +_{\mathcal{H}} k_3{:}t = k_2{:}(\iota_{k_2}(r) +_{k_2} s) +_{\mathcal{H}} k_3{:}t = k_2{:}((\iota_{k_2}(r) +_{k_2} s) +_{k_2} t)$. By associativity of $+_{k_2}$ we have $k_2{:}((\iota_{k_2}(r) +_{k_2} s) +_{k_2} t) = k_2{:}(\iota_{k_2}(r) +_{k_2} (s +_{k_2} t)) = k_1{:}r +_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t)$. If $k_3 \neq k_2$ and $k_3 \neq \mathsf{Nat}$ then we have $(k_1{:}r +_{\mathcal{H}} k_2{:}s) +_{\mathcal{H}} k_3{:}t = k_2{:}((\iota_{k_2}(r) +_{k_2} s) +_{\mathcal{H}} k_3{:}t = \mathsf{Triv}{:}\infty$. We also have $k_1{:}r +_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = k_1{:}r +_{\mathcal{H}} \mathsf{Triv}{:}\infty = \mathsf{Triv}{:}\infty$. If $k_3 = \mathsf{Nat}$ then $(k_1{:}r +_{\mathcal{H}} k_2{:}s) +_{\mathcal{H}} k_3{:}t = k_2{:}(\iota_{k_2}(r) +_{k_2} s) +_{\mathcal{H}} k_3{:}t = k_2{:}((\iota_{k_2}(r) +_{k_2} s) +_{k_2} \iota_{k_2}(t)) = k_2{:}(\iota_{k_2}(r) +_{k_2} (s +_{k_2} \iota_{k_2}(t))) = k_1{:}r +_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t)$.

($+_{\mathcal{H}}4$) $k_2 = \mathsf{Nat}, k_1 \neq \mathsf{Nat}$
Analogous to the case above.

Monotonicity. If $k_1{:}r \preceq_{\mathcal{H}} k_2{:}s$ and $k_3{:}t \preceq_{\mathcal{H}} k_4{:}u$ then $k_1{:}r +_{\mathcal{H}} k_3{:}t \preceq_{\mathcal{H}} k_2{:}s +_{\mathcal{H}} k_4{:}u$. We split cases on the definition of $k_1{:}r \preceq_{\mathcal{H}} k_2{:}s$.

($\preceq_{\mathcal{H}}$ 1) $k_1 = k_2$ and $k_1, k_2 \neq \mathsf{Triv}$
If $k_2 = k_3 = k_4$ we have by ($\preceq_{\mathcal{H}}$ 1) $r \preceq_{k_1} s$ and $t \preceq_{k_1} u$, so we have the thesis by the monotonicity of $+_{k_1}$ with respect to $\preceq_{k_1}$ and ($\preceq_{\mathcal{H}}$ 1). If $k_3 = k_4$ and $k_2 \neq k_4$ then we have $k_2{:}s +_{\mathcal{H}} k_4{:}u = \infty$, so, by ($\preceq_{\mathcal{H}}$ 2) we have the thesis. If $k_4 = \mathsf{Triv}$ by ($\preceq_{\mathcal{H}}$ 2) and since $k_2{:}s +_{\mathcal{H}} k_4{:}u = \infty$ we have the thesis. If $k_3 = \mathsf{Nat}$ and $k_4 \neq \mathsf{Nat}$ and $k_4 = k_1$ then, by ($\preceq_{\mathcal{H}}$ 1) we have $r \preceq_{k_2} s$ and $\iota_{k_2}(t) \preceq_{k_2} u$. By the monotonicity of $+_{k_2}$ with respect to $\preceq_{k_2}$ we have $r +_{k_2} s \preceq_{k_2} \iota_{k_2}(t) +_{k_2} u$ and so, by ($\preceq_{\mathcal{H}}$ 1), ($+_{\mathcal{H}}1$) and ($+_{\mathcal{H}}4$), we have the thesis. If $k_3 = \mathsf{Nat}$ and $k_4 \neq \mathsf{Nat}$ and $k_4 \neq k_1$ then by $k_2{:}s +_{\mathcal{H}} k_4{:}u = \infty$ and ($\preceq_{\mathcal{H}}$ 2) we have the thesis.

($\preceq_{\mathcal{H}}$ 2) $k_2 = \mathsf{Triv}{:}\infty$
Since $k_2{:}s +_{\mathcal{H}} k_4{:}u = \infty$ by ($\preceq_{\mathcal{H}}$ 2) we have the thesis.

($\preceq_{\mathcal{H}}$ 3) $k_1 = \mathsf{Nat}$ and $k_2 \neq \mathsf{Nat}$
If $k_1 = k_3 = \mathsf{Nat}$ and $k_4 = k_2$ then we have $\iota_{k_2}(r) \preceq_{k_2} s$ and $\iota_{k_2}(t) \preceq_{k_2} u$, so by monotonicity of $+_{k_2}$ with respect to $\preceq_{k_2}$ we have $\iota_{k_2}(r) +_{k_2} s \preceq_{k_2} t +_{k_2} u$. By ($\preceq_{\mathcal{H}}$ 1), ($+_{\mathcal{H}}4$) and ($+_{\mathcal{H}}1$) we have the thesis. The other cases are analogous as the cases in which $k_3 = \mathsf{Nat}$ and $k_4 \neq \mathsf{Nat}$.

Identity element. $k{:}r +_{\mathcal{H}} \mathsf{Nat}{:}0 = \mathsf{Nat}{:}0 +_{\mathcal{H}} k{:}r = k{:}r$ for all $k{:}r \in |\mathcal{H}|$. If $k = \mathsf{Nat}$ then $\mathsf{Nat}{:}0 +_{\mathcal{H}} k{:}r = \mathsf{Nat}{:}0 +_{\mathcal{H}} \mathsf{Nat}{:}r = \mathsf{Nat}{:}(\mathbf{0} +_{\mathsf{Nat}} r) = \mathsf{Nat}{:}r = k{:}r$. If $k \neq \mathsf{Nat}$ then $\mathsf{Nat}{:}0 +_{\mathcal{H}} k{:}r = k{:}(\iota_k(\mathbf{0}) +_k r) = k{:}r$ since $\iota_k(\mathbf{0}) = \mathbf{0}_k$ and $\mathbf{0}_k +_k r = r$. □

Proof of Lemma 4.7. We have to prove the same properties as Lemma 4.6, except commutativity. Since $\cdot_{\mathcal{H}}$ is defined similarly to $+_{\mathcal{H}}$ we can ignore the cases already covered in the previous proof and consider only the additional cases of $\cdot_{\mathcal{H}}$:

Associativity. $(k_1{:}r \cdot_{\mathcal{H}} k_2{:}s) \cdot_{\mathcal{H}} k_3{:}t = k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s \cdot_{\mathcal{H}} k_3{:}t)$. We consider only one possible definition of $k_1{:}r \cdot_{\mathcal{H}} k_2{:}s$:

$(\cdot_{\mathcal{H}}5)$ $k_1{:}r = \mathsf{Nat}{:}0$ or $k_2{:}s = \mathsf{Nat}{:}0$ or $k_3{:}t = \mathsf{Nat}{:}0$
We consider only $k_1{:}r = \mathsf{Nat}{:}0$, the other cases are similar. We have $k_1{:}r \cdot_{\mathcal{H}} k_2{:}s = \mathsf{Nat}{:}0$ by $(\cdot_{\mathcal{H}}5)$. Again by $(\cdot_{\mathcal{H}}5)$ we have $\mathsf{Nat}{:}0 \cdot_{\mathcal{H}} k_3{:}t = \mathsf{Nat}{:}0$. Since $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s \cdot_{\mathcal{H}} k_3{:}t) = \mathsf{Nat}{:}0$ for all k_2, k_3, s, t we have the thesis.

Monotonicity. If $k_1{:}r \preceq_{\mathcal{H}} k_2{:}s$ and $k_3{:}t \preceq_{\mathcal{H}} k_4{:}u$ then $k_1{:}r \cdot_{\mathcal{H}} k_3{:}t \preceq_{\mathcal{H}} k_2{:}s \cdot_{\mathcal{H}} k_4{:}u$. The only interesting case is when $k_1{:}r = \mathsf{Nat}{:}0$ or $k_3{:}t = \mathsf{Nat}{:}0$. We consider only the first case, the other is analogous. We have $k_2{:}s \cdot_{\mathcal{H}} k_4{:}u = k'{:}r'$ for a given r', k'. We also have that $\mathbf{0}_{k'} \preceq_{k'} r'$ and $\iota_{k'}(\mathbf{0}) = \mathbf{0}_{k'}$, so, by $(\cdot_{\mathcal{H}}5)$ and $(\preceq_{\mathcal{H}}3)$ we derive $k_1{:}r \cdot_{\mathcal{H}} k_3{:}t = \mathsf{Nat}{:}0 \preceq_{\mathcal{H}} k_2{:}s \cdot_{\mathcal{H}} k_4{:}u$, that is, the thesis.

Identity element. $k{:}r \cdot_{\mathcal{H}} \mathsf{Nat}{:}1 = \mathsf{Nat}{:}1 \cdot_{\mathcal{H}} k{:}r = k{:}r$ for all $k{:}r \in |\mathcal{H}|$. The proof is analogous to the $+_{\mathcal{H}}$ case. □

Proof of Theorem 4.8. We have to prove the properties listed in Definition 4.1. We already proved that $\langle |\mathcal{H}|, \preceq_{\mathcal{H}} \rangle$ is a partially ordered set with binary joins $\vee_{\mathcal{H}}$, that $\langle |\mathcal{H}|, \preceq_{\mathcal{H}}, +_{\mathcal{H}}, \mathsf{Nat}{:}0 \rangle$ is a partially ordered commutative monoid and that $\langle |\mathcal{H}|, \preceq_{\mathcal{H}}, \cdot_{\mathcal{H}}, \mathsf{Nat}{:}1 \rangle$ is a partially ordered monoid. It has remained to prove:

Distributivity. $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = k_1{:}r \cdot_{\mathcal{H}} k_2{:}s +_{\mathcal{H}} k_1{:}r \cdot_{\mathcal{H}} k_3{:}t$ and $(k_2{:}s +_{\mathcal{H}} k_3{:}t) \cdot_{\mathcal{H}} k_1{:}r = k_2{:}s \cdot_{\mathcal{H}} k_1{:}r +_{\mathcal{H}} k_3{:}t \cdot_{\mathcal{H}} k_1{:}r$, for all $k_1{:}r, k_2{:}s, k_3{:}t \in |\mathcal{H}|$. We prove only left-distributivity, right-distributivity is analogous. We split cases on the definition of $k_2{:}s +_{\mathcal{H}} k_3{:}t$.

$(+_{\mathcal{H}}1)$ $k_2 = k_3$
If $k_1 = k_2$ and $k_1{:}r, k_2{:}(s +_{k_2} t) \neq \mathsf{Nat}{:}0$ we have the thesis since $\cdot_{k_1}$ distributes over $+_{k_1}$. If $k_1 \neq k_2$ and $k_1, k_2 \neq \mathsf{Nat}$ we have $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = k_1{:}r \cdot_{\mathcal{H}} k_2{:}(s +_{k_2} t) = \mathsf{Triv}{:}\infty$ and $k_1{:}r \cdot_{\mathcal{H}} k_2{:}s +_{\mathcal{H}} k_1{:}r \cdot_{\mathcal{H}} k_3{:}t = \mathsf{Triv}{:}\infty +_{\mathcal{H}} \mathsf{Triv}{:}\infty = \mathsf{Triv}{:}\infty$. If $k_1 = \mathsf{Nat}$, $r \neq \mathbf{0}$ and $k_2 \neq \mathsf{Nat}$ we have $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = k_1{:}r \cdot_{\mathcal{H}} k_2{:}(s +_{k_2} t) = k_2{:}(\iota_{k_2}(r) \cdot_{k_2} (s +_{k_2} t))$. We have $k_2{:}(\iota_{k_2}(r) \cdot_{k_2} (s +_{k_2} t)) = k_2{:}((\iota_{k_2}(r) \cdot_{k_2} s) +_{k_2} (\iota_{k_2}(r) \cdot_{k_2} t)) = k_2{:}(\iota_{k_2}(r) \cdot_{k_2} s) +_{\mathcal{H}} k_2{:}(\iota_{k_2}(r) \cdot_{k_2} t) = k_1{:}r \cdot_{\mathcal{H}} k_2{:}s +_{\mathcal{H}} k_1{:}r \cdot_{\mathcal{H}} k_3{:}t$. If $k_2 = \mathsf{Nat}$, $s, t \neq \mathbf{0}$ and $k_1 \neq \mathsf{Nat}$, the proof is analogous. If $k_1{:}r = \mathsf{Nat}{:}0$ we have $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = \mathsf{Nat}{:}0$ by $(\cdot_{\mathcal{H}}5)$ and $k_1{:}r \cdot_{\mathcal{H}} k_2{:}s +_{\mathcal{H}} k_1{:}r \cdot_{\mathcal{H}} k_3{:}t = \mathsf{Nat}{:}0 +_{\mathcal{H}} \mathsf{Nat}{:}0 = \mathsf{Nat}{:}0$. If $k_2{:}(s +_{k_2} t) = \mathsf{Nat}{:}0$ we know that $k_2{:}s = \mathsf{Nat}{:}0$ and $k_3{:}t = \mathsf{Nat}{:}0$. We also have $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = \mathsf{Nat}{:}0$ by $(\cdot_{\mathcal{H}}5)$ and $k_1{:}r \cdot_{\mathcal{H}} k_2{:}s +_{\mathcal{H}} k_1{:}r \cdot_{\mathcal{H}} k_3{:}t = \mathsf{Nat}{:}0 +_{\mathcal{H}} \mathsf{Nat}{:}0 = \mathsf{Nat}{:}0$.

$(+_{\mathcal{H}}2)$ $k_2 \neq k_3$ and $k_2, k_3 \neq \mathsf{Nat}$
If $k_1{:}r \neq \mathsf{Nat}{:}0$ we have $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = k_1{:}r \cdot_{\mathcal{H}} \mathsf{Triv}{:}\infty = \mathsf{Triv}{:}\infty$. We know $k_1{:}r \cdot_{\mathcal{H}} k_2{:}s +_{\mathcal{H}} k_1{:}r \cdot_{\mathcal{H}} k_3{:}t = k'{:}s' +_{\mathcal{H}} k''{:}t'$. We know that necessarily $k' \neq k''$ and $k', k'' \neq \mathsf{Nat}$, so $k'{:}s' +_{\mathcal{H}} k''{:}t' = \mathsf{Triv}{:}\infty$. If $k_1{:}r = \mathsf{Nat}{:}0$ we have $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = \mathsf{Nat}{:}0$ and $k_1{:}r \cdot_{\mathcal{H}} k_2{:}s +_{\mathcal{H}} k_1{:}r \cdot_{\mathcal{H}} k_3{:}t = \mathsf{Nat}{:}0 +_{\mathcal{H}} \mathsf{Nat}{:}0 = \mathsf{Nat}{:}0$.

$(+_{\mathcal{H}}3)$ $k_2 \neq \mathsf{Nat}$ and $k_3 = \mathsf{Nat}$
if $k_1 = k_2$ we have $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = k_1{:}(r \cdot_{k_1} (s +_{k_1} \iota_{k_1}(t)))$ and so by $(\cdot_{\mathcal{H}}1)$ and $(+_{\mathcal{H}}1)$ and since $\cdot_{k_1}$ and $+_{k_1}$ have the required property, we have the thesis. If $k_1 = \mathsf{Nat}$ and $r \neq \mathbf{0}$, then $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = k_2{:}(\iota_{k_2}(r) \cdot_{k_2} (s +_{k_2} \iota_{k_2}(t)))$. By $(\cdot_{\mathcal{H}}1)$, $(+_{\mathcal{H}}1)$, $(\cdot_{\mathcal{H}}3)$, $(+_{\mathcal{H}}3)$ and since $\cdot_{k_2}$ and $+_{k_2}$ have the required property, we have the thesis. If $k_1{:}r = \mathsf{Nat}{:}0$ by $(\cdot_{\mathcal{H}}5)$ we have $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = \mathsf{Nat}{:}0$ and by $(\cdot_{\mathcal{H}}5)$ we have $k_1{:}r \cdot_{\mathcal{H}} k_2{:}s +_{\mathcal{H}} k_1{:}r \cdot_{\mathcal{H}} k_3{:}t = \mathsf{Nat}{:}0 +_{\mathcal{H}} \mathsf{Nat}{:}0 = \mathsf{Nat}{:}0$, that is, the thesis. If $k_1 \neq k_2$ and $k_1 \neq \mathsf{Nat}$ we have $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s +_{\mathcal{H}} k_3{:}t) = k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s')$. By $(\cdot_{\mathcal{H}}2)$ we have $k_1{:}r \cdot_{\mathcal{H}} (k_2{:}s') = \mathsf{Triv}{:}\infty$. We also known by $(\cdot_{\mathcal{H}}2)$ that $k_1{:}r \cdot_{\mathcal{H}} k_2{:}s +_{\mathcal{H}} k_1{:}r \cdot_{\mathcal{H}} k_3{:}t = \mathsf{Triv}{:}\infty +_{\mathcal{H}} \mathsf{Triv}{:}\infty = \mathsf{Triv}{:}\infty$.

$(+_{\mathcal{H}}4)$ $k_2 = \mathsf{Nat}$ and $k_3 \neq \mathsf{Nat}$
Similar to the case above.

Zero element. $k{:}r \cdot \mathsf{Nat}{:}0 = \mathsf{Nat}{:}0 \cdot k{:}r = \mathsf{Nat}{:}0$, for all $k{:}r \in |\mathcal{H}|$. By definition in $(\cdot_{\mathcal{H}}5)$.

$\mathsf{Nat}{:}0$ **is minimum element.** $\mathsf{Nat}{:}0 \preceq_{\mathcal{H}} k{:}r$ for all $k{:}r \in |\mathcal{H}|$. We know that for all kinds k, for all its elements r, it holds $\mathbf{0}_k \preceq_k r$. We know that $\iota_k(\mathbf{0}) = \mathbf{0}_k$ so by $(\preceq_{\mathcal{H}}3)$ we have $\mathsf{Nat}{:}0 \preceq_{\mathcal{H}} k{:}r$. □

References

[1] Andreas Abel, Jean-Philippe Bernardy, A unified view of modalities in type systems, Proc. ACM Program. Lang. 90 (2020) 1.

[2] Robert Atkey, Syntax and semantics of quantitative type theory, in: Anuj Dawar, Erich Grädel (Eds.), IEEE Symposium on Logic in Computer Science, LICS 2018, ACM, 2018, pp. 56–65.

[3] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, Arnaud Spiwack, Linear Haskell: practical linearity in a higher-order polymorphic language, Proc. ACM Program. Lang. 2(POPL) (2018) 5:1–5:29.

[4] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, A Java-like calculus with user-defined coeffects, in: Ugo Dal Lago, Daniele Gorla (Eds.), ICTCS'22 - Italian Conf. on Theoretical Computer Science, in: CEUR Workshop Proceedings, vol. 3284, 2022, pp. 66–78, CEUR-WS.org.

[5] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, Multi-graded featherweight Java, in: European Conference on Object-Oriented Programming, ECOOP 2023, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, in press.
[6] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, Marco Servetto, Coeffects for sharing and mutation, Proc. ACM Program. Lang. 6(OOPSLA2) (2022) 870–898.
[7] Edwin C. Brady, Idris 2: quantitative type theory in practice, in: Anders Møller, Manu Sridharan (Eds.), European Conference on Object-Oriented Programming, ECOOP 2021, in: LIPIcs, vol. 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 9:1–9:26.
[8] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, Steve Zdancewic, A core quantitative coeffect calculus, in: Zhong Shao (Ed.), European Symposium on Programming, ESOP 2013, in: Lecture Notes in Computer Science, vol. 8410, Springer, 2014, pp. 351–370.
[9] Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, Stephanie Weirich, A graded dependent type system with a usage-aware semantics, in: Proceedings of ACM on Programming Languages, 5(POPL), 2021, pp. 1–32.
[10] Ugo Dal Lago, Francesco Gavazzo, A relational theory of effects and coeffects, Proc. ACM Program. Lang. 6(POPL) (2022) 1–28.
[11] Ornela Dardha, Elena Giachino, Davide Sangiorgi, Session types revisited, in: Danny De Schreye, Gerda Janssens, Andy King (Eds.), PPDP'12, ACM, 2012, pp. 139–150.
[12] Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, Tarmo Uustalu, Combining effects and coeffects via grading, in: Jacques Garrigue, Gabriele Keller, Eijiro Sumii (Eds.), ACM International Conference on Functional Programming, ICFP 2016, ACM, 2016, pp. 476–489.
[13] Dan R. Ghica, Alex I. Smith, Bounded linear types in a resource semiring, in: Zhong Shao (Ed.), European Symposium on Programming, ESOP 2013, in: Lecture Notes in Computer Science, vol. 8410, Springer, 2014, pp. 331–350.
[14] Colin S. Gordon, Designing with static capabilities and effects: use, mention, and invariants (pearl), in: Robert Hirschfeld, Tobias Pape (Eds.), European Conference on Object-Oriented Programming, ECOOP 2020, in: LIPIcs, vol. 166, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 10:1–10:25.
[15] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, Joe Duffy, Uniqueness and reference immutability for safe parallelism, in: Gary T. Leavens, Matthew B. Dwyer (Eds.), ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 2012, ACM, 2012, pp. 21–40.
[16] Philipp Haller, Martin Odersky, Capabilities for uniqueness and borrowing, in: Theo D'Hondt (Ed.), European Conference on Object-Oriented Programming, ECOOP 2010, in: Lecture Notes in Computer Science, vol. 6183, Springer, 2010, pp. 354–378.
[17] Atsushi Igarashi, Benjamin C. Pierce, Philip Wadler, Featherweight Java: a minimal core calculus for Java and GJ, in: ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999, ACM, 1999, pp. 132–146.
[18] Conor McBride, I got plenty o' nuttin', in: Sam Lindley, Conor McBride, Philip W. Trinder, Donald Sannella (Eds.), A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, in: Lecture Notes in Computer Science, vol. 9600, Springer, 2016, pp. 207–233.
[19] Dominic Orchard, Vilem-Benjamin Liepelt, Harley Eades III, Quantitative program reasoning with graded modal types, in: Proceedings of ACM on Programming Languages, vol. 110, 2019, p. 1.
[20] Tomas Petricek, Dominic A. Orchard, Alan Mycroft Coeffects, Unified static analysis of context-dependence, in: Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, David Peleg (Eds.), Automata, Languages and Programming, ICALP 2013, in: Lecture Notes in Computer Science, vol. 7966, Springer, 2013, pp. 385–397.
[21] Tomas Petricek, Dominic A. Orchard, Alan Mycroft, Coeffects: a calculus of context-dependent computation, in: Johan Jeuring, Manuel M.T. Chakravarty (Eds.), ACM International Conference on Functional Programming, ICFP 2014, ACM, 2014, pp. 123–135.
[22] James Wood, Robert Atkey, A framework for substructural type systems, in: Ilya Sergey (Ed.), European Symposium on Programming, ESOP 2022, in: Lecture Notes in Computer Science, vol. 13240, Springer, 2022, pp. 376–402.