A Java-like Calculus with User-Defined Coeffects

Riccardo Bianchini[1,*], Francesco Dagnino[1], Paola Giannini[2] and Elena Zucca[1]

[1]University of Genova, Italy
[2]University of Piemonte Orientale, Italy

Abstract

We propose a Java-like calculus where declared variables can be annotated by *coeffects* specifying constraints on their use, such as linearity or privacy levels. Annotations are written in the language itself, as expressions of type `Coeffect`, a predefined class which can be extended by user-defined subclasses, modeling the coeffects desired for a specific application. We formalize the type system and prove subject reduction, which includes preservation of coeffects, and show several examples.

Keywords

type systems, operational semantics, Java-like languages

1. Introduction

Type-and-coeffect systems [1, 2, 3, 4, 5, 6, 7, 8] are type systems where the typing judgment takes an enriched form, such as $x_1 :_{c_1} C_1, \ldots, x_n :_{c_n} C_n \vdash e : C$, with $c_1, \ldots, c_n$ *coeffects* modeling how the corresponding variables are used in e.

For instance, coeffects of shape $c ::= 0 \mid 1 \mid \omega$ trace when a variable is either not used, or used linearly (that is, exactly once), or used in an unrestricted way, respectively, in the expression e. In this way, functions, e.g., $\lambda x{:}\mathsf{int}.5$, $\lambda x{:}\mathsf{int}.x$, and $\lambda x{:}\mathsf{int}.x + x$, which have the same type in the simply-typed lambda calculus, can be distinguished by adding coeffect annotations: $\lambda x{:}\mathsf{int}[0].5$, $\lambda x{:}\mathsf{int}[1].x$, and $\lambda x{:}\mathsf{int}[\omega].x + x$. Other typical examples are exact usage (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. Moreover, some kind of order relation is often required as well.

This approach has been exploited to a fully-fledged programming language in Granule [6], 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 exact usage, security levels, intervals, infinity, and products of coeffects; however, the available coeffects are fixed once and for all.

In this paper, we investigate the possibility of providing a similar support in Java-like languages, by allowing the programmer to write coeffect annotations in variable declarations.

Proceedings of the 23rd Italian Conference on Theoretical Computer Science, Rome, Italy, September 7-9, 2022
*Corresponding author.
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)
0000-0003-0491-7652 (R. Bianchini); 0000-0003-3599-3535 (F. Dagnino); 0000-0003-2239-9529 (P. Giannini); 0000-0002-6833-6470 (E. Zucca)

Moreover, we rely on the inheritance mechanism of OO languages to allow the programmer to write her/his own coeffects. More in detail, coeffect annotations are expressions of (subclasses of) a predefined class `Coeffect`, analogously to Java exceptions which are expressions of (subclasses of) `Exception`.

In Sect. 2 we define the underlying calculus, and in Sect. 3 the type-and-coeffect system, stating its soundness. In Sect. 4 we show some examples. We discuss further work in Sect. 5.

2. Calculus

The syntax and operational semantics of the language are shown in Fig. 1. We write es as metavariable for $e_1, \ldots, e_n$, $n \geq 0$, and analogously for other sequences. We assume sets of *variables* $x, y, z, \ldots$, including the special variable `this`, *class names* C, *field names* f, and *method names* m. Besides the standard predefined class `Object`, root of the inheritance hierarchy, we assume another predefined class `Coeffect`, whose definition will be given in the next section. A subclass of `Coeffect` is called a *coeffect class*.

$$
\begin{array}{lcll}
e & ::= & x \mid e.f \mid \texttt{new}\ C(es) \mid e.m(es) \mid \{C[\hat{v}]\ x = e;\ e'\} & \text{expression} \\
 & & \mid \texttt{case}\ e\ \texttt{of}\ (C_1[\hat{v}_1]\ x_1)e_1\ (C_2[\hat{v}_2]\ x_2)e_2 & \\
v & ::= & \texttt{new}\ C(vs) & \text{value} \\
\mathcal{E} & ::= & [\,] \mid \mathcal{E}.f \mid \texttt{new}\ C(vs, \mathcal{E}, es) \mid \mathcal{E}.m(es) \mid v.m(vs, \mathcal{E}, es) & \text{evaluation context} \\
 & & \mid \{C[\hat{v}]\ x = \mathcal{E};\ e\} \mid \texttt{case}\ \mathcal{E}\ \texttt{of}\ (C_1[\hat{v}_1]\ x_1)e_1\ (C_2[\hat{v}_2]\ x_2)e_2 &
\end{array}
$$

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

$$
\text{(FIELD-ACCESS)}\ \frac{}{\texttt{new}\ C(v_1, \ldots, v_n).f_i \to v_i}\ \begin{array}{l}\mathsf{fields}(C) = C_1\ f_1; \ldots C_n\ f_n; \\ i \in 1..n\end{array}
$$

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

$$
\text{(BLOCK)}\ \frac{}{\{C[\hat{v}]\ x = v;\ e\} \to e[v/x]}
$$

$$
\text{(CASE-1)}\ \frac{}{\texttt{case}\ \texttt{new}\ C(vs)\ \texttt{of}\ (C_1[\hat{v}_1]\ x_1)e_1\ (C_2[\hat{v}_2]\ x_2)e_2 \to e_1[\texttt{new}\ C(vs)/x_1]}\ C \leq C_1
$$

$$
\text{(CASE-2)}\ \frac{}{\texttt{case}\ \texttt{new}\ C(vs)\ \texttt{of}\ (C_1[\hat{v}_1]\ x_1)e_1\ (C_2[\hat{v}_2]\ x_2)e_2 \to e_2[\texttt{new}\ C(vs)/x_2]}\ \begin{array}{l}C \not\leq C_1 \\ C \leq C_2\end{array}
$$

Figure 1: Calculus

In addition to standard constructs (variable, field access, constructor and method invocation, and block) we include a *typecase* construct, since in coeffect classes a check of the runtime type will be typically needed to write binary methods, see the examples in Sect. 4. In the calculus,

we choose this compact construct, essentially the same used in [9], to keep the syntax minimal; it could be encoded in true Java by using conditional, `instanceof`, and cast.

A block expression consists of the declaration of a local variable, and the body in which this variable can be used. A variable declaration specifies a type (a class name C), an initialization expression e, and a *coeffect annotation* $\hat{v}$, which is syntactically a value[1]; however, we use the meta-variable $\hat{v}$ rather than v to suggest that it is expected to be a value of a coeffect class. The syntax of coeffect annotations is inspired by that used in Granule [6].

A typecase expression consists of an initial expression e, and two alternatives, each one specifying a local variable with its type and coeffect annotation, and an expression. Expression e_1 is executed if the result of the initial expression has a subtype of C_1, with x_1 bound to such result; otherwise, expression e_2 is executed, with x_2 bound to the result, provided that it has a subtype of C_2. This second check is guaranteed to always succeed by the type system.

Reduction rules in Fig. 1 are straightforward. 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, the inherited first
- $\mathsf{mbody}(C, m)$ gives, for each method m of class C, parameters and body.

3. Type System

The type system uses *coeffects* $c ::= n \mid \hat{v}$, with $n \in \mathbb{N}$, and $\hat{v}$ *user-defined coeffect*, that is, value expected to be of a coeffect class.

The typing judgment has shape $\Gamma \vdash e : C$, where Γ is a (type-and-coeffect) context, that is, a (finite) map from variables to pairs of a coeffect and a type (class name), written $\Gamma = x_1 :_{c_1} C_1, \ldots, x_n :_{c_n} C_n$. Equivalently, Γ can be seen as the pair of a *coeffect context* and a *type context*, maps from variables to coeffects and types, respectively, with the same domain.

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. We first define the operations on contexts, then we show the typing rules, and finally we define how the assumed operations on coeffects are derived from user-defined methods in coeffect classes.

The operations on contexts are the following:

Sup
$$\emptyset \vee \Gamma = \Gamma$$
$$(x :_c C, \Gamma) \vee \Delta = x :_c C, (\Gamma \vee \Delta) \text{ if } x \notin \mathsf{dom}(\Delta)$$
$$(x :_c C, \Gamma) \vee (x :_{c'} C, \Delta) = x :_{c \vee c'} C, (\Gamma \vee \Delta)$$

Partial order
$$(x :_c C, \Gamma) \leq (x :_{c'} C, \Delta) \text{ if } c' \leq c \text{ and } \Gamma \leq \Delta$$

Sum
$$\emptyset + \Gamma = \Gamma \qquad (x :_c C, \Gamma) + \Delta = x :_c C, (\Gamma + \Delta) \text{ if } x \notin \mathsf{dom}(\Delta)$$
$$(x :_c C, \Gamma) + (x :_{c'} C, \Delta) = x :_{c+c'} C, (\Gamma + \Delta)$$

[1] Coeffect annotations could be generalized to be arbitrary expressions; here we use this simpler assumption to make the presentation lighter.

$$\text{(T-SUB)}\ \frac{\Gamma \vdash e : C' \quad C' \leq C}{\Gamma \vdash e : C} \qquad \text{(T-VAR)}\ \frac{}{(0{\times}\Gamma) + x :_1 C \vdash x : C}$$

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

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

$$\text{(T-INVK)}\ \frac{\Gamma_0 \vdash e_0 : C \qquad \Gamma_i \vdash e_i : C_i \quad \forall i \in 1..n}{(c_0'{\times}\Gamma_0) + \ldots + (c_n'{\times}\Gamma_n) \vdash e_0.m(e_1, \ldots, e_n) : C} \quad \mathsf{mtype}(C, m) = \hat{v}_0, C_1^{\hat{v}_1} \ldots C_n^{\hat{v}_n} \to C \quad c_i' = \hat{v}_i \vee 1 \quad \forall i \in 0..n$$

$$\text{(T-BLOCK)}\ \frac{\begin{array}{l}\emptyset \vdash \hat{v} : \texttt{Coeffect} \\ \Gamma \vdash e : C \qquad \Gamma', x :_c C \vdash e' : C'\end{array} \quad c \leq \hat{v}}{(c'{\times}\Gamma) + \Gamma' \vdash \{C[\hat{v}]\ x = e\,;\ e'\} : C'} \quad c' = \hat{v} \vee 1$$

$$\text{(T-CASE)}\ \frac{\begin{array}{l}\emptyset \vdash \hat{v}_i : \texttt{Coeffect} \quad \forall i \in 1..2 \\ \Gamma \vdash e : C \\ \Gamma_i, x :_{c_i} C_i \vdash e_i : C' \quad \forall i \in 1..2\end{array}}{(c_1' \vee c_2'){\times}\Gamma + (\Gamma_1 \vee \Gamma_2) \vdash \texttt{case}\ e\ \texttt{of}\ (C_1[\hat{v}_1]\ x_1) e_1\ (C_2[\hat{v}_2]\ x_2) e_2 : C'} \quad \begin{array}{l} C \not\leq C_1 \text{ implies } C \leq C_2 \\ c_i \leq \hat{v}_i \quad \forall i \in 1..2 \\ c_i' = \hat{v}_i \vee 1 \quad \forall i \in 1..2 \end{array}$$

Figure 2: Coeffect system with user defined coeffects

Scalar multiplication $\qquad c{\times}\emptyset = \emptyset \qquad c{\times}(x :_{c'} C, \Gamma) = x :_{c \times c'} C, (c{\times}\Gamma)$

As the reader can note, operations on coeffect contexts are obtained by pointwise application of the corresponding operations on coeffects, and then they are lifted to type-and-coeffect contexts. In this lifting, sum becomes partial since a variable in the domain of both contexts is required to have the same type.

The typing rules are given in Fig. 2. In the subsumption rule (T-SUB), the type can be made more general, as usual. The context could, symmetrically, be made more specific ($\Gamma \leq \Gamma'$), by pointwise overapproximating coeffects; however, to stress that they can be actually computed bottom-up, we prefer a presentation where overapproximation of coeffects is an explicit side condition of the rules where it is needed, see rules (T-BLOCK) and (T-CASE).

In rule (T-VAR), the given variable is used exactly once. 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 sup (greatest lower bound) of the coeffect annotation of the corresponding parameter, and the one coeffect. The sup with the one coeffect guarantees to take into account the coeffects of the initialization expression for parameters not used in the body, see the last part of Example 2 in Sect. 4. This is customary in type-and-coeffect systems for call-by-value calculi. The rule uses the auxiliary function mtype which returns an enriched method type, where the types of the parameters and of `this` have coeffect annotations. Such coeffect annotations should have a coeffect class as type. Formally, if $\mathsf{mtype}(C, m) = \hat{v}_0, C_1^{\hat{v}_1} \ldots C_n^{\hat{v}_n} \to C$, then $\emptyset \vdash \hat{v}_i :$ `Coeffect` for $i \in 0..n$. Moreover, $\mathsf{mtype}(C, m)$ and $\mathsf{mbody}(C, m)$ should be either both

undefined or both defined with the same number of parameters; in the latter case, the method body should be well-typed with respect to the method type, notably by typechecking the method body we should get coeffects which are (overapproximated by) those specified in the annotations. Formally, if $\mathsf{mbody}(C, m) = (x_1 \dots x_n, e)$, and $\mathsf{mtype}(C, m) = \hat{v}_0, C_1^{\hat{v}_1} \dots C_n^{\hat{v}_n} \to C$, the following condition must hold:

$$\text{(T-METH)} \qquad \texttt{this} :_{c_0} C, x_1 :_{c_1} C_1, \dots, x_n :_{c_n} C_n \vdash e : C \qquad c_i \leq \hat{v}_i \quad \forall i \in 0..n$$

In rule (T-BLOCK), the coeffects of the initialization expression are multiplied by the sup of the coeffect annotation of the variable, and the one coeffect, and then summed with those of the body. Analogously to method invocation, the sup with the one coeffect is needed when the variable is not used in the body. The side condition $c \leq \hat{v}$ checks that the variable is used in the body accordingly with the annotation $\hat{v}$ written by the programmer. The premise $\emptyset \vdash \hat{v} : \texttt{Coeffect}$ checks that $\hat{v}$ is a user-defined coeffect. In rule (T-CASE), coeffects are handled analogously as in rule (T-BLOCK). The first side-condition ensures that the cases are exhaustive.

We describe now how operations on coeffects are derived from user-defined methods in the coeffect classes. The predefined class `Coeffect` is defined as follows:

```
class Coeffect {
  Coeffect sup(Coeffect c) { new Coeffect() }
  Coeffect sum(Coeffect c) { new Coeffect() }
  Coeffect mult(Coeffect c) { new Coeffect() }
  Coeffect zero() { new Coeffect() }
  Coeffect one() { new Coeffect() }
}
```

Being subclasses of `Coeffect`, coeffect classes implement the above methods, corresponding to the expected operations on coeffects: sup, sum, multiplication, zero, and one. In the class `Coeffect`, such operations are those of the trivial semiring consisting of only one element.

Let $\mathsf{nf}(e)$ denote the normal form of e. The normal form, if any, is unique, since it is easy to see that the reduction relation is deterministic. Moreover, we assume that methods `sup`, `sum`, `mult`, `zero`, and `one` in coeffect classes always terminate, so that normal forms in the definitions below are defined.

Recall that coeffects are either natural numbers or user-defined coeffects (values of a coeffect class). Natural numbers are internally used[2] by the type system, notably in rule (T-VAR) which is independent from coeffect classes in the program. As a consequence, operations on coeffects derived from user-defined methods need to handle natural numbers as well. This is achieved through a translation into user-defined coeffects, which corresponds to the unique homomorphism from the initial semiring to another one. That is, $\texttt{in}^{\hat{v}}(n)$ is the user-defined coeffect corresponding to n in the class of $\hat{v}$, defined by:

$$\texttt{in}^{\hat{v}}(0) = \mathsf{nf}(\hat{v}\texttt{.zero()}) \qquad \texttt{in}^{\hat{v}}(1) = \mathsf{nf}(\hat{v}\texttt{.one()})$$
$$\texttt{in}^{\hat{v}}(n+1) = \mathsf{nf}(\texttt{in}^{\hat{v}}(n)\texttt{.sum(}\hat{v}\texttt{.one())}) \text{ for } n \geq 1$$

We can now define operations on coeffects. To avoid confusion, standard operations on natural numbers have the explicit subscript $\mathbb{N}$.

[2]Derivations in Example 2 illustrate their interplay with user-defined coeffects.

Sup
$$\hat{v}\vee n = \mathsf{nf}(\hat{v}\,.\,\mathtt{sup}(\mathtt{in}^{\hat{v}}(n))) \qquad n\vee\hat{v} = \mathsf{nf}(\mathtt{in}^{\hat{v}}(n)\,.\,\mathtt{sup}(\hat{v}))$$
$$\hat{v}_1\vee\hat{v}_2 = \mathsf{nf}(\hat{v}_1\,.\,\mathtt{sup}(\hat{v}_2)) \qquad n_1\vee n_2 = \max_{\mathbb{N}}(n_1, n_2)$$

Sum
$$\hat{v} + n = \mathsf{nf}(\hat{v}\,.\,\mathtt{sum}(\mathtt{in}^{\hat{v}}(n))) \qquad n + \hat{v} = \mathsf{nf}(\mathtt{in}^{\hat{v}}(n)\,.\,\mathtt{sum}(\hat{v}))$$
$$\hat{v}_1 + \hat{v}_2 = \mathsf{nf}(\hat{v}_1\,.\,\mathtt{sum}(\hat{v}_2)) \qquad n_1 + n_2 = n_1 +_{\mathbb{N}} n_2$$

Multiplication
$$\hat{v} \times n = \mathsf{nf}(\hat{v}\,.\,\mathtt{mul}(\mathtt{in}^{\hat{v}}(n))) \qquad n \times \hat{v} = \mathsf{nf}(\mathtt{in}^{\hat{v}}(n)\,.\,\mathtt{mul}(\hat{v}))$$
$$\hat{v}_1 \times \hat{v}_2 = \mathsf{nf}(\hat{v}_1\,.\,\mathtt{mul}(\hat{v}_2)) \qquad n_1 \times n_2 = n_1 \times_{\mathbb{N}} n_2$$

Note that the order $\leq$ on coeffects can be derived from the $\vee$ operator, as follows: $c_1 \leq c_2$ iff $c_1 \vee c_2 = c_2$. We prefer to implement the sup operator, rather than the order, in coeffect classes, since the latter would require to include the primitive type `boolean` in the calculus.

Our main technical result is subject reduction (Theorem 1), expressing, as customary in type-and-coeffect systems, that not only types, but also coeffects are preserved by reduction. As expected, to prove this property we need, as in proofs in the literature, assumptions on operations derived from user-defined methods, corresponding to have a semiring (1-8) with an additional order-theoretic structure (9-15). Ensuring such properties is in charge of the programmer, possibily using some verification tool.

Assumptions 1. *For all user-defined coeffects* $\hat{v}_1, \hat{v}_2, \hat{v}_3$*:*

1. $\hat{v}_1 + \hat{v}_2 = \hat{v}_2 + \hat{v}_1$
2. $\hat{v}_1 + (\hat{v}_2 + \hat{v}_3) = (\hat{v}_1 + \hat{v}_2) + \hat{v}_3$
3. $\hat{v}_1 + 0 = \hat{v}_1$
4. $\hat{v}_1 \times 1 = 1 \times \hat{v}_1 = \hat{v}_1$
5. $\hat{v}_1 \times 0 = 0 \times \hat{v}_1 = 0$
6. $\hat{v}_1 \times (\hat{v}_2 \times \hat{v}_3) = (\hat{v}_1 \times \hat{v}_2) \times \hat{v}_3$
7. $\hat{v}_1 \times (\hat{v}_2 + \hat{v}_3) = \hat{v}_1 \times \hat{v}_2 + \hat{v}_1 \times \hat{v}_3$
8. $(\hat{v}_1 + \hat{v}_2) \times \hat{v}_3 = \hat{v}_1 \times \hat{v}_3 + \hat{v}_2 \times \hat{v}_3$
9. $\hat{v}_1 \vee \hat{v}_1 = \hat{v}_1$
10. $\hat{v}_1 \vee (\hat{v}_2 \vee \hat{v}_3) = (\hat{v}_1 \vee \hat{v}_2) \vee \hat{v}_3$
11. $\hat{v}_1 \vee \hat{v}_2 = \hat{v}_2 \vee \hat{v}_1$
12. $\hat{v}_1 \vee (\hat{v}_1 + \hat{v}_2) = (\hat{v}_1 + \hat{v}_2)$
13. $\hat{v}_1 + (\hat{v}_2 \vee \hat{v}_3) = (\hat{v}_1 + \hat{v}_2) \vee (\hat{v}_1 + \hat{v}_3)$
14. $\hat{v}_1 \times (\hat{v}_2 \vee \hat{v}_3) = (\hat{v}_1 \times \hat{v}_2) \vee (\hat{v}_1 \times \hat{v}_3)$
15. $(\hat{v}_2 \vee \hat{v}_3) \times \hat{v}_1 = (\hat{v}_2 \times \hat{v}_1) \vee (\hat{v}_3 \times \hat{v}_1)$

The proof of Theorem 1 uses the standard lemmas of inversion for expressions and contexts (Lemmas 1 and 2), and substitution in evaluation contexts and terms (Lemmas 3 and 4).

Lemma 1 (Inversion).

1. *If* $\Gamma \vdash x : C$*, then* $\Gamma = (0 \times \Gamma') + x :_1 C$ *for some* Γ'*.*
2. *If* $\Gamma \vdash e\,.f_i : C$*, then we have* $\Gamma \vdash e : C$ *and* $\mathsf{fields}(C) = C_1\, f_1\,; \ldots C_n\, f_n\,;$ *and* $C_i = C$*.*
3. *If* $\Gamma \vdash \mathtt{new}\, C(e_1, \ldots, e_n) : C$*, then we have* $\mathsf{fields}(C) = C_1\, f_1\,; \ldots C_n\, f_n\,;, \Gamma_i \vdash e_i : C_i$ *for all* $i \in 1..n$ *and* $\Gamma = \Gamma_1 + \ldots + \Gamma_n$*.*
4. *If* $\Gamma \vdash e_0.m(e_1, \ldots, e_n) : C$*, then we have* $\Gamma_0 \vdash e_0 : C$*,* $\mathsf{mtype}(C, m) = \hat{v}_0, C_1^{\hat{v}_1} \ldots C_n^{\hat{v}_n} \to C$*,* $c_i' = \hat{v}_i \vee 1$ *for all* $i \in 0..n$*,* $\Gamma_i \vdash e_i : C_i$ *for all* $i \in 1..n$ *and* $\Gamma = (c_0' \times \Gamma_0) + (c_1' \times \Gamma_1) + \ldots + (c_n' \times \Gamma_n)$*.*
5. *If* $\Gamma \vdash \{C[\hat{v}]\ x = e\,;\ e'\} : C'$*, then we have* $\emptyset \vdash \hat{v} : \mathtt{Coeffect}$ *and* $\Gamma = c' \times \Delta' + \Gamma'$ *where* $c' = \hat{v} \vee 1$*,* $c \leq \hat{v}$ *and* $\Delta' \vdash e : C$ *and* $\Gamma', x :_c C' \vdash e' : C'$*.*

6. *If* $\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ (C_1[\hat{v}_1]\ x_1) e_1\ (C_2[\hat{v}_2]\ x_2) e_2 : C$ *then we have* $\Gamma = (c_1' \vee c_2') \times \Gamma' + (\Gamma_1 \vee \Gamma_2)$, $\emptyset \vdash \hat{v}_i : \texttt{Coeffect}$ *for* $i \in 1..2$, $\Gamma' \vdash e : C$, $\Gamma_i, x :_{c_i} C_i \vdash e_i : C'$ *for* $i \in 1..2$, $C \not\leq C_1$ *implies* $C \leq C_2$, $c_i \leq \hat{v}_i$ *for* $i \in 1..2$ *and* $c_i' = \hat{v}_i \vee 1$ *for* $i \in 1..2$.

Lemma 2 (Context Inversion). *If* $\Gamma \vdash \mathcal{E}[e] : C$, *then* $\Gamma' + x :_c C' \vdash \mathcal{E}[x] : C$ *and* $\Delta \vdash e : C'$ *for some* Γ', Δ, $x \notin \mathsf{dom}(\Gamma)$, c *and* C' *such that* $\Gamma = \Gamma' + c \times \Delta$.

Proof 1. *By induction on* $\mathcal{E}$.

Lemma 3 (Context Substitution). *If* $\Delta \vdash e : C'$ *and* $\Gamma + x :_c C' \vdash \mathcal{E}[x] : C$, *with* $x \notin \mathsf{dom}(\Gamma)$, *then* $\Gamma + c \times \Delta \vdash \mathcal{E}[e] : C$.

Proof 2. *By induction on the derivation of* $\Gamma + x :_{\hat{v}} C \vdash \mathcal{E}[x] : C'$.

Lemma 4 (Substitution). *If* $\Delta \vdash e' : C'$ *and* $\Gamma, x :_c C' \vdash e : T$ *then* $\Gamma + c \times \Delta \vdash e[e'/x] : C$.

Theorem 1. *If* $\Gamma \vdash e : C$ *and* $e \to e'$, *then* $\Gamma' \vdash e' : C$ *such that* $\Gamma \leq \Gamma'$.

Proof 3. *By induction on the reduction relation. We show some significant cases.*

(CTX) *By Lemma 2 we have* $\Gamma' + x :_{\hat{v}} C' \vdash \mathcal{E}[x] : C$, $\Delta \vdash e : C'$ *and* $\Gamma = \Gamma' + \hat{v} \times \Delta$. *By induction hypothesis we derive* $\Delta \vdash e' : C'$. *By Lemma 3 we derive* $\Gamma' + \hat{v} \times \Delta \vdash \mathcal{E}[e'] : C$, *that is,* $\Gamma \vdash \mathcal{E}[e'] : C$.

(FIELD-ACCESS) *By Lemma 1(2) we have* $\Gamma \vdash \texttt{new}\ C(v_1, \ldots, v_n) : C$ *and* $\mathsf{fields}(C) = C_1\ f_1\ ; \ldots C_n\ f_n\ ;$ *with* $C_i = C$. *By Lemma 1(3) we have* $\mathsf{fields}(C) = C_1\ f_1\ ; \ldots C_n\ f_n\ ;$, $\Gamma_j \vdash \hat{v}_j : C_j$ *for all* $j \in 1..n$ *and* $\Gamma = \Gamma_1 + \ldots + \Gamma_n$. *We know that* $i \in 1...n$, $\hat{v}_i = e'$ *and* $\Gamma_i \vdash e' : C$. *By Assumptions (12) and (10) we know* $\Gamma \leq \Gamma_i$, *so we have the thesis .*

(BLOCK) *By Lemma 1(5) we have* $\vdash \hat{v} : \texttt{Coeffect}$, $\Gamma = \Gamma' + c' \times \Delta'$ *where* $c' = \hat{v} \vee 1$, $c \leq \hat{v}$, $\Delta' \vdash e : C$ *and* $\Gamma', x :_c C' \vdash e' : C'$. *Applying Lemma 4 we derive* $\Gamma' + c' \times \Delta' \vdash e_2[e_1/x] : C'$. *Applying Assumption (14) we derive the thesis.*

4. Examples

We use as syntactic sugar `case` expressions with more than two alternatives, which can be easily encoded in the calculus. Moreover, variable declarations with no coeffect annotation are assumed to have a `new Coeffect()` coeffect. Some other simplifications in code are explained when used. Coeffect annotations are emphasized in grey to help the reader.

Example 1. Linearity As first example we show how to implement the classical coeffect system mentioned in the Introduction, which traces when a variable is either not used (coeffect 0), or used linearly (that is, exactly once, coeffect 1), or used in an unrestricted way (coeffect ω).

```
class Linearity extends Coeffect{
  Coeffect zero(){ new Zero()}
  Coeffect one(){new One()}
}

class Zero extends Linearity{
  Coeffect sup(Coeffect c) {
    case c of
      (Zero x) x
      (Linearity x) new Omega()
      (Coeffect x) new Coeffect()
  }
}
  Coeffect sum(Coeffect c) {
    case c of
      (Linearity x) x
      (Coeffect x) new Coeffect()
  }
  Coeffect mult(Coeffect c) {
    case c of
      (Linearity x) new Zero()
      (Coeffect x) new Coeffect()
  }
}

class Omega extends Linearity{
  Coeffect sup(Coeffect c) {
    case c of
      (Linearity x) new Omega()
      (Coeffect x) new Coeffect()
  }
  Coeffect sum(Coeffect c) {
    case c of
      (Linearity x) new Omega()
      (Coeffect x) new Coeffect()
  }
```

```
class One extends Linearity{
  Coeffect sup(Coeffect c) {
    case c of
      (One x) new One()
      (Linearity x) new Omega()
      (Coeffect x) new Coeffect()}
  }
  Coeffect sum(Coeffect c) {
    case c of
      (Zero x) new One()
      (Linearity x) new Omega()
      (Coeffect x) new Coeffect()
  }
  Coeffect mult(Coeffect c) {
    case c of
      (Linearity x) x
      (Coeffect x) new Coeffect()
  }
}
```

```
  Coeffect mult(Coeffect c) {
    case c of
      (Zero x) new Zero()
      (Linearity x) new Omega()
      (Coeffect x) new Coeffect()
  }
}
```

Example 2. Using linearity coeffects After declaring the above coeffect classes, the programmer can use them in annotations, as in the simple example below, where they are used for the annotation of `this`, written between the method name and the list of parameters.

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

Let us see some examples of how the type system works. 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 5. A type derivation for the method body is as follows:

$$\dfrac{\dfrac{}{\texttt{this} :_1 \texttt{A} \vdash \texttt{this} : \texttt{A}}\,(\text{T-VAR}) \qquad \dfrac{}{\texttt{this} :_1 \texttt{A} \vdash \texttt{this} : \texttt{A}}\,(\text{T-VAR})}{\texttt{this} :_2 \texttt{A} \vdash \texttt{new Pair(this, this)} : \texttt{Pair}}\,(\text{T-NEW})$$

Set $\omega = \texttt{new Omega()}$. The condition (T-METH) requires $\texttt{this} :_\omega \texttt{A} \leq \texttt{this} :_2 \texttt{A}$, and this holds since we can derive $2 \leq \omega$ from

- $2 \vee \omega = \mathsf{nf}(\texttt{in}^\omega(2).\texttt{sup}(\omega))$
- $\texttt{in}^\omega(2) = \mathsf{nf}((\omega.\texttt{zero}().\texttt{sum}(\omega.\texttt{one}())).\texttt{sum}(\omega.\texttt{one}())) = \omega$
- $\mathsf{nf}(\omega.\texttt{sup}(\omega)) = \omega$

If the expression new `Pair(this,this).fst` was the body of method `identity`, instead, then the method would be ill-typed. Indeed, we would have a similar derivation:

$$\dfrac{\dfrac{\dfrac{\text{(T-VAR)}\ \overline{\texttt{this} :_1 \texttt{A} \vdash \texttt{this} : \texttt{A}} \quad \text{(T-VAR)}\ \overline{\texttt{this} :_1 \texttt{A} \vdash \texttt{this} : \texttt{A}}}{\text{(T-NEW)}\ \texttt{this} :_2 \texttt{A} \vdash \texttt{new Pair(this,this)} : \texttt{Pair}}}{\text{(T-FIELD-ACCESS)}\ \texttt{this} :_2 \texttt{A} \vdash \texttt{new Pair(this,this).fst} : \texttt{A}}$$

However, $\texttt{this} :_{\texttt{new One()}} \texttt{A} \leq \texttt{this} :_2 \texttt{A}$ does not hold, since $2 \not\leq \texttt{new One()}$:

- $2 \vee \texttt{new One()} = \mathsf{nf}(\texttt{in}^{\texttt{new One()}}(2).\texttt{sup}(\texttt{new One()}))$
- $\texttt{in}^{\texttt{new One()}}(2) =$
  $\mathsf{nf}((\texttt{new One().zero().sum(new One().one())}).\texttt{sum}(\texttt{new One().one()})) = \omega$
- $\mathsf{nf}(\omega.\texttt{sup}(\texttt{new One()})) = \omega$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{(T-VAR)}\ \overline{x :_1 \texttt{A} \vdash x : \texttt{A}} \quad \text{(T-VAR)}\ \overline{y :_1 \texttt{A} \vdash y : \texttt{A}}}{\text{(T-NEW)}\ x :_1 \texttt{A}, y :_1 \texttt{A} \vdash \texttt{new Pair}(x,y) : \texttt{Pair}}}{\text{(T-FIELD-ACCESS)}\ x :_1 \texttt{A}, y :_1 \texttt{A} \vdash \texttt{new Pair}(x,y)\texttt{.fst} : \texttt{A}}}{\text{(T-INVK)}\ x :_\omega \texttt{A}, y :_\omega \texttt{A} \vdash \texttt{new Pair}(x,y)\texttt{.fst.duplicate()} : \texttt{Pair}}$$

where $\mathsf{mtype}(\texttt{A}, \texttt{dup}) = \omega, \epsilon \to \texttt{Pair}$, $\omega = \omega \vee 1$, and $\omega = \omega \times 1$.

We can see that natural numbers are translated into user-defined linearity coeffects when coeffect operations occur in the derivation.

Finally, we show an example motivating that in rules (T-INVK), (T-BLOCK), and (T-CASE) the sup with 1 is needed. For instance, in this way we can derive the judgment $y :_\omega \texttt{A} \vdash \{\texttt{Pair}[\texttt{new Zero()}]\ x = y.\texttt{duplicate}; \texttt{new A()}\} : \texttt{A}$.

Example 3. Product of coeffects The following example shows that, besides single coeffect classes, the programmer can also define general constructions. The class `Product` implements coeffects which are the *product* of two arbitrary coeffects.

```
class Product extends Coeffect{ Coeffect left; Coeffect right;
  Coeffect one(){new Product(left.one(),right.one())}
  Coeffect zero(){new Product(left.zero(),right.zero())}
  Coeffect sup(Coeffect c) {
    case c of
      (Product p) new Product(this.left.sup(p.left),this.right.sup(p.right))
      (Coeffect x) new Coeffect()
  }
```

```
  Coeffect sum(Coeffect c) {
    case c of
      (Product p) new Product(this.left.sum(p.left),this.right.sum(p.right))
      (Coeffect x) new Coeffect()
  }
  Coeffect mult(Coeffect c) {
    case c of
      (Product p) new Product(this.left.mul(p.left),this.right.mul(p.right))
      (Coeffect x) new Coeffect()
  }
}
```

Example 4. Sessions Finally, we show a more significant programming example, which also uses the coeffect type `Level`, providing 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`$\leq$`Private`$\leq$`Public`. Sum is the sup and product is defined by $c_1 \times c_2 =$`Irrelevant` if either $c_1 =$`Irrelevant` or $c_2 =$`Irrelevant`, otherwise $c_1 \times c_2 = c_1 \vee c_2$. It is easy to write the `Level` classes which implement the previous definitions.

The example illustrates how binary sessions could be implemented using our coeffect annotations. We take inspiration from the encoding of sessions into the π-calculus with variants and linear I/O types of [10]. We assume to have the following classes implementing linear input and output channels over which we can send a message and, if needed, some private data.

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

class InPrivChannel{
  (Msg,OptData,OutPrivChannel) rcv [new One()] (OutPrivChannel [new One()] cont) {...}
}
class Unit{ }
class OptData{ }
class Some extends OptData{A inf;}
class None extends OptData{ }
class Msg { }
class NextData extends Msg{ }
class Stop extends Msg{ }
class OK extends Msg{ }
class KO extends Msg{ }
```

The linearity of the channel is expressed by annotating the receiver of the send and receive methods with the `One` coeffect of the `Linearity` coeffects. The `send` method of the class `OutPrivChannel` takes as input, in addition to the message and the data to be sent, an input channel that will be used, by whoever is sending the message, to wait for the answer to the sent message. On the other side, the `rcv` method of class `InPrivChannel` returns a message, a data and an output channel[3] that will be used, by whoever received the message, to continue the conversation. The

[3]To simplify the code we allow the method to return a triple here.

message does not have a coeffect as this is irrelevant.

The class `Server` that follows implements a server which waits for a message that can be either `NextData` or `Stop`. In the first case, the server sends to whoever sent the message the message OK, processes the data, and goes back to waiting for a message. In the second case, it stops returning OK (we use this message also to signal that the protocol ended successfully). If it receives any other message, the server stops returning KO, meaning failure of the exchange.

```
class Server {
  Msg main(InPrivChannel [new One()] c){
    (Msg msg,OptData [new Private()] d,OutPrivChannel [new One()] c1)=c.rcv();
      case msg of
        (NextData [new Zero()] y){
          OutPrivChannel [new One()] outCh= new OutPrivChannel();
          InPrivChannel  [new One()] inCh= new InPrivChannel();
          c1.send(msg,new None(),outCh)
          main(inCh)
        }
        (Stop [new Zero()] y) new OK()
        (Msg [new Zero()] y) new KO()
 }
}
```

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. A client of the previous server sends a data to the server and then, if the response of the server is OK, either sends a `NextData` or a `Stop` message. If the response is not OK the client returns KO, meaning failure of the exchange.

```
class Client {
  Msg main(OutPrivChannel [new One()] c){
    OutPrivChannel [new One()] outCh=new OutPrivChannel();
    InPrivChannel  [new One()] inCh=new InPrivChannel();
    c.send(new nextData(),new Some(new A()),outCh){
    (Msg msg,Data [new Private()] d,OutPrivChannel [new One()] c1)=inCh.rcv();
    case msg of
      (OK [new Zero()] y){c1.send(Stop,new None(),new InPrivChannel()); new OK()}
      (Msg [new Zero()] y) new KO()
  }
}
```

The client above sends just one data, however clients could interact with the server sending more data. As for the server, the initial message of the client is sent on the channel which is the parameter of the method, and then a new pair of channels is generated and used for the successive interaction.

The computation of the server and the client is started by sending an output channel to the client and an input channel to the server. The fact that the channels are linear ensures that they will be used to realise the wanted binary session. Assuming to have a parallel composition operator, the server and the client can be started as follows.

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

5. Conclusion

We proposed a Java-like calculus supporting, in variable declarations, coeffect annotations, allowing to express how such variables should be used. Thanks to the inheritance mechanism of OO languages, programmers are able to define their own coeffect annotations. Indeed, they are expressions of (subclasses of) `Coeffect`, a predefined class which can be extended by user-defined subclasses, similarly to Java exceptions which are expressions of (subclasses of) `Exception`. We formally defined the type system and proved subject reduction, which includes preservation of coeffects, and provided several examples.

There are many interesting directions for further work. The most natural development is adding *graded modal types*. Indeed, a limitation of the current proposal 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). Graded modal types, which are, roughly, types annotated with coeffects (grades) [2, 6, 8], also similar to types annotated with *modifiers* or *capabilities* [11, 12, 13], would allow to overcome this limitation.

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 [1]. In our work, this would correspond to allow a "global" annotation in a method's signature.

Finally, 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, have not been developed yet.

On the more applicative side, we could investigate how the proposal scales to realistic subsets of Java. Implementations could use in a parametric way auxiliary tools, notably a termination checker to prevent divergence in methods implementing coeffect operations, and/or a verifier to ensure that they provide the required properties (assumptions at page 6).

Acknowledgments

The authors thank the anonymous referees who provided useful and detailed comments. This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X).

References

[1] T. Petricek, D. A. Orchard, A. Mycroft, Coeffects: a calculus of context-dependent computation, in: J. Jeuring, M. M. T. Chakravarty (Eds.), ACM International Conference on Functional Programming, ICFP 2014, ACM Press, 2014, pp. 123–135. doi:10.1145/2628136.2628160.

[2] A. Brunel, M. Gaboardi, D. Mazza, S. Zdancewic, A core quantitative coeffect calculus, in: Z. Shao (Ed.), European Symposium on Programming, ESOP 2013, volume 8410 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 351–370. doi:10.1007/978-3-642-54833-8_19.

[3] R. Atkey, Syntax and semantics of quantitative type theory, in: A. Dawar, E. Grädel (Eds.), IEEE Symposium on Logic in Computer Science, LICS 2018, ACM Press, 2018, pp. 56–65. doi:10.1145/3209108.3209189.

[4] M. Gaboardi, S. Katsumata, D. A. Orchard, F. Breuvart, T. Uustalu, Combining effects and coeffects via grading, in: J. Garrigue, G. Keller, E. Sumii (Eds.), ACM International Conference on Functional Programming, ICFP 2016, ACM Press, 2016, pp. 476–489. doi:10.1145/2951913.2951939.

[5] D. R. Ghica, A. I. Smith, Bounded linear types in a resource semiring, in: Z. Shao (Ed.), European Symposium on Programming, ESOP 2013, volume 8410 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 331–350. doi:10.1007/978-3-642-54833-8_18.

[6] D. Orchard, V. Liepelt, H. E. III, Quantitative program reasoning with graded modal types, Proceedings of ACM on Programming Languages 3 (2019) 110:1–110:30. doi:10.1145/3341714.

[7] P. Choudhury, H. E. III, R. A. Eisenberg, S. Weirich, A graded dependent type system with a usage-aware semantics, Proceedings of ACM on Programming Languages 5 (2021) 1–32. doi:10.1145/3434331.

[8] U. Dal Lago, F. Gavazzo, A relational theory of effects and coeffects, Proceedings of ACM on Programming Languages 6 (2022) 1–28. doi:10.1145/3498692.

[9] A. Igarashi, H. Nagira, Union types for object-oriented programming, J. Object Technol. 6 (2007) 47–68. doi:10.5381/jot.2007.6.2.a3.

[10] O. Dardha, E. Giachino, D. Sangiorgi, Session types revisited, in: D. D. Schreye, G. Janssens, A. King (Eds.), PPDP'12, ACM, 2012, pp. 139–150. doi:10.1145/2370776.2370794.

[11] P. Haller, M. Odersky, Capabilities for uniqueness and borrowing, in: T. D'Hondt (Ed.), European Conference on Object-Oriented Programming, ECOOP 2010, volume 6183 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 354–378.

[12] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, J. Duffy, Uniqueness and reference immutability for safe parallelism, in: G. T. Leavens, M. B. Dwyer (Eds.), ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 2012, ACM Press, 2012, pp. 21–40.

[13] C. S. Gordon, Designing with static capabilities and effects: Use, mention, and invariants (pearl), in: R. Hirschfeld, T. Pape (Eds.), European Conference on Object-Oriented Programming, ECOOP 2020, volume 166 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 10:1–10:25. doi:10.4230/LIPIcs.ECOOP.2020.10.