Multi-Graded Featherweight Java

Riccardo Bianchini
DIBRIS, University of Genova, Italy

Francesco Dagnino
DIBRIS, University of Genova, Italy

Paola Giannini
DiSSTE, University of Eastern Piedmont, Vercelli, Italy

Elena Zucca
DIBRIS, University of Genova, Italy

Abstract

Resource-aware type systems statically approximate not only the expected result type of a program, but also the way external resources are used, e.g., how many times the value of a variable is needed. We extend the type system of Featherweight Java to be resource-aware, parametrically on an arbitrary *grade algebra* modeling a specific usage of resources. We prove that this type system is *sound* with respect to a resource-aware version of reduction, that is, a well-typed program has a reduction sequence which does not get stuck due to resource consumption. Moreover, we show that the available grades can be *heterogeneous*, that is, obtained by combining grades of different kinds, via a minimal collection of homomorphisms from one kind to another. Finally, we show how grade algebras and homomorphisms can be specified as Java classes, so that grade annotations in types can be written in the language itself.

2012 ACM Subject Classification Theory of computation → Type structures

Keywords and phrases Graded modal types, Java

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.3

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

Acknowledgements We thank the anonymous referees for their useful suggestions.

1 Introduction

Recently, a considerable amount of research [25, 7, 2, 14, 15, 23, 8, 11] has been devoted to type systems allowing reasoning about resource usage. In *(type-and-)coeffect systems*, the typing judgment takes the shape $x_1 :_{r_1} T_1, \ldots, x_n :_{r_n} T_n \vdash e : T$, where the *coeffect* (*grade*) r_i models how variable x_i is 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 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 examples are exact usage (coeffects are natural numbers), and privacy levels. *Graded modal types* go further, by decorating types themselves with grades, in order to specify how *the result of an expression* should be used. In the different proposals in literature, grades have a similar algebraic structure, basically a semiring specifying *sum* $+$, *multiplication* $\cdot$, and 0 and 1 constants, and some kind of order relation. Here, we will assume a variant of this notion called *grade algebra*.

Resource-aware typing has been exploited in a fully-fledged programming language in Granule [23], a functional language equipped with graded modal types, hence allowing the programmer to write function declarations similar to those above. In Granule, different


© Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca;
licensed under Creative Commons License CC-BY 4.0

kinds of coeffects can be used at the same time, including naturals for exact usage, privacy levels, intervals, infinity, and products of coeffects; however, available grades are fixed in the language. The initial objective of the work presented here was to study a similar support for Java-like languages, by introducing, in a variant of Featherweight Java (FJ) [19], types decorated with grades. Moreover, we wanted these grades to be taken, parametrically, in an arbitrary grade algebra; even more, we did not want this grade algebra to be fixed as in Granule, but to be extendable by the programmer with user-defined grades, by relying on the inheritance mechanism of OO languages. In the quest for such goals, we came up with several ideas which are novel, to our knowledge, with respect to the literature on resource-aware type systems, as detailed in the outline of contributions given below.

Resource-aware parametric FJ reduction. Given a resource-aware type system, we would like to prove that typing overapproximates the use of resources. However, resource usage is not modeled in standard operational semantics; for this reason, [8] proposed an instrumented operational semantics[1] and proved a soundness theorem showing correct accounting of resource usage. Inspired by this work, we define a *resource-aware semantics* for FJ, parametric on an arbitrary grade algebra, which tracks how much each available resource is consumed at each step, and is stuck when the needed amount of a resource is not available. Differently from [8], the semantics is given *independently* from the type system, as is the standard approach in calculi. That is, the aim is also to provide a simple purely semantic model which takes into account usage of resources. The resource-aware reduction is sound with respect to the standard reduction, but clearly not complete, since a reduction step allowed in the standard semantics could be impossible due to resource consumption.

Graded FJ. After defining the resource-aware calculus, we define the resource-aware type system. That is, types are decorated with grades, allowing the programmer to specify how a variable, a field or the result of a method should be used, e.g., how many times. Our approach is novel with respect to that generally used in the literature on graded modal types. Notably, in such works the production of types is $T ::= \ldots \mid T^r$, that is, grade decorations can be arbitrarily nested. Correspondingly, the syntax includes an explicit *box* construct, which transforms a term of type T into a term of type T^r, through a *promotion* rule which multiplies the context with r, and a corresponding unboxing mechanism. Here, we prefer a much lighter approach, likely more convenient for Java-like languages, where the syntax of terms is not affected. The production for types is $T ::= C^r$, that is, all types (here only class names) are (once) graded; in contexts, types are non-graded, and grades are used as coeffects, leading to a judgment of shape $x_1 :_{r_1} C_1, \ldots, x_n :_{r_n} C_n \vdash e : C^r$. Finally, since there is no boxing/unboxing, there is no explicit promotion rule, but different grades can be assigned to an expression, assuming different coeffect contexts. We prove a soundness theorem, stating that the graded type system overapproximates resource usage, hence guaranteeing soundness, and, as a consequence, completeness with respect to standard reduction for well-typed programs.

Combining grades. The next matter is how to make the language *multi-graded*, in the sense that the programmer can use grades of different kinds, e.g., both natural numbers and privacy levels. This poses the problem of defining the result when grades of different kinds should

[1] Subsequently the model of [8] was used, in [21], to trace reference counting for uniqueness.

be combined by the type system. This issue has been considered in the Granule language [23], where, however, the available kinds of grades are fixed, hence can be combined in an ad-hoc way. We would like to have much more flexibility, that is, to allow the programmer to define grades to be added to those already available, very much in the same way a Java programmer can define her/his own class of exceptions. To this end, we define a construction which, given a family of grade algebras and a family of homomorphisms, leads to a unique grade algebra of *heterogeneous grades*. This allows a modular approach, in the sense that the developed meta-theory, including the proof of results, applies to this case as well.

Grades as Java expressions. Finally, we consider the issue of providing linguistic support to specify the desired grade algebras and homomorphisms. Of course this could be done by using an ad-hoc configuration language. However, we believe an interesting solution is that the grade annotations could be written themselves in Java, again analogously to what happens with exceptions. We describe how Java classes corresponding to grade algebras and homomorphisms could be written, providing some examples.

A preliminary step towards the results described in the current paper is [3], which proposes a first version of the type system with only coeffects (types are not graded), and a rudimentary version of the construction described above where combining coeffects of different kinds leads to the trivial coeffect.

In Section 2 we formally define grade algebras and related notions. In Section 3 we define the parametric resource-aware reduction for FJ, and in Section 4 the parametric resource-aware type system, proving its soundness. Section 5 defines the construction of the grade algebra of heterogeneous grades, and Section 6 illustrates how to express grade algebras and homomorphisms in Java. Finally, Section 7 surveys related work and Section 8 summarizes the contributions, and outlines future work. Omitted proofs can be found in [4].

2 Algebraic preliminaries

In this section we introduce the algebraic structures we will use throughout the paper. The core of our work is *grades*, namely, annotations in the code expressing how or how much resources are used by the program. As we will see, we need some operations to properly combine grades in the resource-aware semantics and in the typing rules, hence we will assume grades to form an algebraic structure called *grade algebra* defined below.

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

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

and 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* $r, s, t \in |R|$*;*
- $r \cdot \mathbf{0} = \mathbf{0}$ *and* $\mathbf{0} \cdot r = \mathbf{0}$*, for all* $r \in |R|$*;*
- *if* $r \preceq r'$ *and* $s \preceq s'$ *then* $r + s \preceq r' + s'$ *and* $r \cdot s \preceq r' \cdot s'$*, for all* $r, r's, s' \in |R|$*;*
- $\mathbf{0} \preceq r$*, for all* $r \in |R|$*.*

Essentially, a grade algebra is an ordered semiring, that is, a semiring together with a partial order relation on its carrier which makes addition and multiplication monotonic with respect to it. We further require the zero of the semiring to be the least element of the partial order. Our definition is a slight variant of others proposed in literature [7, 15, 22, 2, 14, 1, 23, 8, 27]. 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*.

▶ **Example 2.**

1. The semiring $\mathsf{Nat} = \langle \mathbb{N}, \leq, +, \cdot, 0, 1 \rangle$ of natural numbers with the natural order and usual arithmetic operations is a grade algebra.
2. The *affinity* grade algebra $\langle \{0, 1, \infty\}, \leq, +, \cdot, 0, 1 \rangle\}$ is obtained from the previous one by identifying all natural numbers greater than 1.
3. The trivial semiring Triv, whose carrier is a singleton set $|\mathsf{Triv}| = \{\infty\}$, the partial order is the equality, addition and multiplication are defined in the trivial way and $\mathbf{0}_{\mathsf{Triv}} = \mathbf{1}_{\mathsf{Triv}} = \infty$, is a grade algebra.
4. The semiring $\mathsf{R}^{\infty}_{\geq 0} = \langle [0, \infty], \leq, +, \cdot, 0, 1 \rangle$ of extended non-negative real numbers with usual order and operations, extended to ∞ in the expected way, is a grade algebra.
5. A distributive lattice $\mathsf{L} = \langle |\mathsf{L}|, \leq, \vee, \wedge, \bot, \top \rangle$, where $\vee$ and $\wedge$ denote join and meet operations and $\bot$ and $\top$ the bottom and the top element, respectively, is a grade algebra.
6. Given grade algebras $R = \langle |R|, \preceq_R, +_R, \cdot_R, \mathbf{0}_R, \mathbf{1}_R \rangle$ and $S = \langle |S|, \preceq_S, +_S, \cdot_S, \mathbf{0}_S, \mathbf{1}_S \rangle$, the *product* $R \times S = \langle \{\langle r, s \rangle \mid r \in |R| \wedge s \in |S|\}, \preceq, +, \cdot, \langle \mathbf{0}_R, \mathbf{0}_S \rangle, \langle \mathbf{1}_R, \mathbf{1}_S \rangle \rangle$, where operations are the pairwise application of the operations for R and S, is a grade algebra.
7. Given a grade algebra $R = \langle |R|, \preceq_R, +_R, \cdot_R, \mathbf{0}_R, \mathbf{1}_R \rangle$, as in [23] we define $\mathsf{Ext}\, R = \langle |R| + \{\infty\}, \preceq, +, \cdot, \mathbf{0}_R, \mathbf{1}_R \rangle$ where $\preceq$ extends $\preceq_R$ by adding $r \preceq \infty$ for all $r \in |\mathsf{Ext}\, R|$ and $+$ and $\cdot$ extend $+_R$ and $\cdot_R$ by $r + \infty = \infty + r = \infty$, for all $r \in |\mathsf{Ext}\, R|$, and $r \cdot \infty = \infty \cdot r = \infty$, for all $r \in |\mathsf{Ext}\, R|$ with $r \neq \mathbf{0}_R$, and $\mathbf{0}_R \cdot \infty = \infty \cdot \mathbf{0}_R = \mathbf{0}_R$. Then, $\mathsf{Ext}\, R$ is a grade algebra.

A homomorphism of grade algebras $f \colon R \to S$ is a monotone function $f \colon \langle |R|, \preceq_R \rangle \to \langle |S|, \preceq_S \rangle$ between the underlying partial orders, which preserves the semiring structure, that is, satisfies the following equations:

- $f(\mathbf{0}_R) = \mathbf{0}_S$ and $f(r +_R s) = f(r) +_S f(s)$, for all $r, s \in |R|$;
- $f(\mathbf{1}_R) = \mathbf{1}_S$ and $f(r \cdot_R s) = f(r) \cdot_S f(s)$, for all $r, s \in |R|$.

Grade algebras and their homomorphisms form a category denoted by $\mathcal{GrAlg}$.

Consider a grade algebra R. Then, we can define functions $\zeta_R \colon |R| \to |\mathsf{Triv}|$ and $\iota_R \colon |\mathsf{Nat}| \to |R|$ as follows:

$$\zeta_R(r) = \infty \qquad \iota_R(m) = \begin{cases} \mathbf{0}_R & \text{if } m = 0 \\ \iota_R(n) +_R \mathbf{1}_R & \text{if } m = n + 1 \end{cases}$$

Roughly, ζ_R maps every element of R to ∞, while ι_R maps a natural number n to the sum in R of n copies of $\mathbf{1}_R$. We can easily check that both these functions give rise to homomorphisms $\zeta_R \colon R \to \mathsf{Triv}$ and $\iota_R \colon \mathsf{Nat} \to R$. This is straightforward for ζ_R, while for ι_R follows by arithmetic induction. Then, we can prove the following result.

▶ **Proposition 3.** *The following facts hold:*

1. Nat *is the initial object in* $\mathcal{GrAlg}$*;*
2. Triv *is the terminal object in* $\mathcal{GrAlg}$*.*

Another kind of objects we will work with are maps assigning grades to variables. These inherit a nice algebraic structure from the one of the underlying grade algebra.

Assume a grade algebra $R = \langle |R|, \preceq, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$ and a set X. The set of functions from X to $|R|$ carries a partially ordered commutative monoid structure given by the pointwise extension of the additive structure of R. That is, given $\gamma, \gamma' \colon X \to |R|$, we define $\gamma \preceq \gamma'$

iff, for all $x \in X$, $\gamma(x) \preceq \gamma'(x)$, and $(\gamma + \gamma')(x) = \gamma(x) + \gamma'(x)$ and $\hat{\mathbf{0}}(x) = \mathbf{0}$, for all $x \in X$. Moreover, we can define a *scalar multiplication*, combining elements of $|R|$ and a function $\gamma\colon X \to |R|$; indeed, we set $(r \cdot \gamma)(x) = r \cdot \gamma(x)$, for all $r \in |R|$ and $x \in X$. It is easy to see that this operation turns the partially ordered commutative monoid of functions from X to $|R|$ into a partially ordered R-module.

The *support* of a function $\gamma\colon X \to |R|$ is the set $\mathsf{S}(\gamma) = \{x \in X \mid \gamma(x) \neq \mathbf{0}\}$. Denote by R^X the set of functions $\gamma\colon X \to |R|$ with finite support. The partial order and operations defined above can be safely restricted to R^X, noting that $\mathsf{S}(\hat{\mathbf{0}}) = \emptyset$, $\mathsf{S}(\gamma + \gamma') \subseteq \mathsf{S}(\gamma) \cup \mathsf{S}(\gamma')$ and $\mathsf{S}(r \cdot \gamma) \subseteq \mathsf{S}(\gamma)$. Therefore, R^X carries a partially ordered R-module structure as well.

As we will see in Section 4, *coeffect contexts* are (representations of) functions in R^X, with X set of variables. The fact that coeffect contexts form a module has been firstly noted in [22, 27], and fully formalized in [5], which also shows a *non-structural* example. That is, a module different from R^X described above, used in the present paper and mostly in the literature, is needed, where operations on coeffect contexts are not pointwise.

3 Resource-aware semantics

Standard operational models do not say anything about resources used by the computation. To address this problem, we follow an approach similar to that in [8], that is, we define an *instrumented* semantics which keeps track of resource usage, hence, in particular, it gets stuck if some needed resource is insufficient. However, unlike [8], the definition of our resource-aware semantics, though parameterized on a grade algebra, is given *independently* of the graded type system, as is the standard approach in calculi; in the next section, we will show how the graded type system actually overapproximates resource usage, hence guarantees soundness. As will be detailed in the following, the resource-aware semantics is non-deterministic, in the sense that, when a resource is needed, it can be consumed in different ways; hence, soundness is *soundness-may*, meaning that there is a reduction which does not get stuck because of standard typing errors or resource consumption.

Reference calculus. The calculus is a variant of FJ [19]. The syntax is reported in the top section of Figure 1. We write *es* as a metavariable for $e_1, \ldots, e_n$, $n \geq 0$, and analogously for other sequences. We assume *variables* $x, y, z, \ldots$, *class names* C, D, *field names* f, and *method names* m. Types are distinct from class names to mean that they could be extended to include other types, e.g., primitive types. In addition to the standard FJ constructs, we have a block expression, consisting of a local variable declaration, and a body.

The semantics is defined differently from the original one; that is, reduction is defined on *configurations* $e|\rho$, where ρ is an *environment*, a finite map from variables into values. In this way, variable occurrences are replaced one at a time by their value in the environment, rather than once and for all. This definition can be easily shown to be equivalent to the original one, and is convenient for our aims since, in this presentation, free variables in an expression can be naturally seen as *resources* which are consumed each time a variable occurrence is *used* (replaced by its value) during execution. In other words, this semantics can be naturally *instrumented* by adding grades expressing the "cost" of resource consumption, as we will do in Figure 2. Apart from that, the rules are straightforward; only note that, in rules (INVK) and (BLOCK), parameters (including `this`) and local variable are renamed to fresh variables, to avoid clashes. Single contextual rules are given, rather than defining evaluation contexts, to be uniform with the instrumented version, where this presentation is more convenient.

e	::=	$x \mid e.f \mid \texttt{new}\ C(es) \mid e.m(es) \mid \{T\,x = e;\ e'\}$	expression
T	::=	C	type (class name)
v	::=	$\texttt{new}\ C(vs)$	value

(VAR) $$\frac{}{x|\rho \to v|\rho}\ \rho(x) = v$$

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

(INVK) $$\frac{}{v_0.m(v_1, \ldots, v_n)|\rho \to e[y_0/\texttt{this}][y_1/x_1 \ldots y_n/x_n]|\rho'}\ \begin{array}{l}v_0 = \texttt{new}\ C(\_) \\ \mathsf{mbody}(C, m) = \langle x_1 \ldots x_n, e\rangle \\ y_0, \ldots, y_n \notin \mathsf{dom}(\rho) \\ \rho' = \rho, y_0 \mapsto v_0, \ldots, y_n \mapsto v_n\end{array}$$

(BLOCK) $$\frac{}{\{C\,x = v;\ e\}|\rho \to e[y/x]|\rho, y \mapsto v}\ y \notin \mathsf{dom}(\rho)$$

(FIELD-ACCESS-CTX) $$\frac{e|\rho \to e'|\rho'}{e.f|\rho \to e'.f|\rho'}$$

(NEW-CTX) $$\frac{e_i|\rho \to e_i'|\rho'}{\texttt{new}\ C(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n)|\rho \to \texttt{new}\ C(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)|\rho'}$$

(INVK-RCV-CTX) $$\frac{e_0|\rho \to e_0'|\rho'}{e_0.m(e_1, \ldots, e_n)|\rho \to e_0'.m(e_1, \ldots, e_n)|\rho'}$$

(INVK-ARG-CTX) $$\frac{e_i|\rho \to e_i'|\rho'}{v_0.m(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n)|\rho \to v_0.m(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)|\rho'}$$

(BLOCK-CTX) $$\frac{e_1|\rho \to e_1'|\rho'}{\{C\,x = e_1;\ e_2\}|\rho \to \{C\,x = e_1';\ e_2\}|\rho'}$$

Figure 1 Syntax and standard reduction.

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

- $\mathsf{fields}(C)$ gives, for each class C, the sequence $T_1\ f_1; \ldots T_n\ f_n;$ of its fields, assumed to have distinct names, with their types;
- $\mathsf{mbody}(C, m)$ gives, for each method m of class C, its parameters and body.

Instrumented reduction. This reduction uses *grades*, ranged over by r, s, t, assumed to form a grade algebra, specifying a *partial order* $\preceq$, a *sum* $+$, a *multiplication* $\cdot$, and constants $\mathbf{0}$ and $\mathbf{1}$, satisfying some axioms, as detailed in Definition 1 of Section 2.

In order to keep track of usage of resources, parametrically on a given grade algebra, we *instrument* reduction as follows.

- The environment associates, to each resource (variable), besides its value, a grade modeling its *allowed usage*.

- Moroever, the reduction relation is *graded*, that is, indexed by a grade r, meaning that it aims at producing a value to be used (at most) r times, or, in more general (non-quantitative) terms, to be used (at most) with grade r.
- The grade of a variable in the environment decreases, each time the variable is used, of the amount specified in the reduction grade[2].
- Of course, this can only happen if the current grade of the variable *can* be reduced of such an amount; otherwise the reduction is stuck.

Before giving the formal definition, we show some simple examples of reductions, considering the grade algebra of naturals of Example 2(1), tracking how many times a resource is used.

▶ **Example 4.** Assume the following classes:

```
class A {}
class Pair {A first; A second}
```

We write $v_{\texttt{Pair}}$ as an abbreviation for $\texttt{new Pair(new A(),new A())}$.

$\{\texttt{A a} = [\texttt{new A()}]_4\texttt{;}\ \{\texttt{Pair p} = [\texttt{new Pair(a,a)}]_2\texttt{;}\ \texttt{new Pair(p.first,p.second)}\}|\emptyset \to_1$
$\{\texttt{Pair p} = [\texttt{new Pair(a,a)}]_2\texttt{;}\ \texttt{new Pair(p.first,p.second)}\}|\texttt{a} \mapsto \langle \texttt{new A()}, 4\rangle \to_1$
$\{\texttt{Pair p} = [\texttt{new Pair(new A(),a)}]_2\texttt{;}\ \texttt{new Pair(p.first,p.second)}\}|\texttt{a} \mapsto \langle \texttt{new A()}, 2\rangle \to_1$
$\{\texttt{Pair p} = [\texttt{new Pair(new A(),new A())}]_2\texttt{;}\ \texttt{new Pair(p.first,p.second)}\}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle \to_1$
$\texttt{new Pair(p.first,p.second)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 2\rangle \to_1$
$\texttt{new Pair(}v_{\texttt{Pair}}\texttt{.first,p.second)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 1\rangle \to_1$
$\texttt{new Pair(new A(),p.second)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 1\rangle \to_1$
$\texttt{new Pair(new A(),}v_{\texttt{Pair}}\texttt{.second)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 0\rangle \to_1$
$v_{\texttt{Pair}}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 0\rangle$

In the example, the top-level reduction is graded 1, meaning that a single value is produced. Subterms are annotated with the grade of their reduction. For instance, the outer initialization expression is annotated 4, meaning that its result can be used (at most) 4 times. To lighten the notation, in this example we omit the index 1. A local variable introduced in a block is added[3] as another available resource in the environment, with the value and the grade of its initialization expression; for instance, the outer local variable is added with grade 4. When evaluating the inner initialization expression, which is reduced with grade 2, each time the variable `a` is used its grade in the environment is decremented by 2.

It is important to notice that the annotations in subterms are *not* type annotations. Except those in arguments of constructor invocation, explained below, annotations are only needed to ensure that reduction of a subterm happens at each step with the same grade, see the formal definition below. We plan to investigate in future work a big-step formulation which would not need such an artifice. In the example above, we have chosen for the reduction of subterms the minimum grade allowing to perform the top-level reduction. We could have chosen any greater grade; instead, with a strictly lower grade, the reduction would be stuck.

As anticipated, in a constructor invocation $\texttt{new}\ C(\texttt{[}e_1\texttt{]}_{r_1}, \ldots, \texttt{[}e_n\texttt{]}_{r_n})$, the annotation r_i plays a special role: intuitively, it specifies that the object to be constructed should contain r_i copies of that field. Formally, this is reflected by the reduction grade of the subterm e_i, which must be exactly $r \cdot r_i$, if r is the reduction grade of the object, specifying how many copies of it the reduction is constructing. Correspondingly, an access to the field can be used (at most) $r \cdot r_i$ times. This is illustrated by the following variant of the previous example.

[2] More precisely, the reduction grade acts as a lower bound for this amount, see comment to rule (VAR).
[3] Modulo renaming to avoid clashes, omitted in the example for simplicity.

▶ **Example 5.** Consider the term

$$\{\texttt{A a} = [\texttt{new A()}]_4\texttt{;}\ \{\texttt{Pair p} = [\texttt{new Pair(a, a)}]_2\texttt{;}\ \texttt{new Pair(}\ [\texttt{p.first}]_2\ \texttt{, p.second)}\}\}$$

As highlighted in grey, the first argument of the constructor invocation which is the body of the inner block is now annotated with 2, meaning that the resulting object should have "two copies" of the field. As a consequence, the expression `p.first` should be reduced with grade 2, as shown below, where $v_{\texttt{Pair}} = \texttt{new Pair(new A(), new A())}$, the first four reduction steps are as in Example 4 and we explicitly write some annotations 1 for clarity

$$\{\texttt{A a} = [\texttt{new A()}]_4\texttt{;}\ \{\texttt{Pair p} = [\texttt{new Pair(}[\texttt{a}]_1\texttt{, a)}]_2\texttt{;}\ \texttt{new Pair(}\ [[\texttt{p}]_1\texttt{.first}]_2\ \texttt{, p.second)}\}\}|\emptyset \to_1^*$$

$$\texttt{new Pair(}\ [[\texttt{p}]_1\texttt{.first}]_2\ \texttt{, p.second)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 2\rangle \to_1$$

$$\texttt{new Pair(}\ [[v_{\texttt{Pair}}]_1\texttt{.first}]_2\ \texttt{, p.second)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 1\rangle \quad \text{STUCK}$$

Reduction of the subterm in grey, aiming at constructing a value (`new A()`) which can be used twice, is stuck, since we cannot obtain two copies of `new A()` from the field `first` of the object $v_{\texttt{Pair}}$. If we choose, instead, to reduce the occurrence of `p` to be used twice, then we get the following reduction, where again we omit steps which are as before:

$$\{\texttt{A a} = [\texttt{new A()}]_4\texttt{;}\ \{\texttt{Pair p} = [\texttt{new Pair(}[\texttt{a}]_1\texttt{, a)}]_2\texttt{;}\ \texttt{new Pair(}\ [[\texttt{p}]_2\texttt{.first}]_2\ \texttt{, p.second)}\}\}|\emptyset \to_1^*$$

$$\texttt{new Pair(}\ [[\texttt{p}]_2\texttt{.first}]_2\ \texttt{, p.second)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 2\rangle \to_1$$

$$\texttt{new Pair(}\ [[v_{\texttt{Pair}}]_2\texttt{.first}]_2\ \texttt{, p.second)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 0\rangle \to_1$$

$$\texttt{new Pair(}[\texttt{new A()}]_2\texttt{, p.second)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 0\rangle \quad \text{STUCK}$$

In this case, the reduction is stuck since we consumed all the available copies of `p` to produce two copies of the field `first`, so now we cannot reduce `p.second`. To obtain a non-stuck reduction, we should choose to reduce the initialization expression of `p` with index 3, hence that of `a` with index 6. To complete the construction of the `Pair`, that is, to get a non-stuck reduction, we should have 3 copies of `p` and therefore 6 copies of `a`.

The formal definition of the instrumented semantics is given in Figure 2. To make the notation lighter, we use the same metavariables of the standard semantics in Figure 1. As explained above, reduction is defined on annotated terms. Notably, in each construct, the subterms which are reduced in contextual rules are annotated, so that their reduction always happens with a fixed grade.

In rule (VAR), which is the key rule where resources are consumed, a variable occurrence is replaced by the associated value in the environment, and its grade s decreases to s', burning a non-zero amount r' of resources which has to be at least the reduction grade. The side condition $r' + s' \preceq s$ ensures that the initial grade of the variable suffices to cover both the consumed grade and the residual grade. To show why the amount of resource consumption should be non-zero, consider, e.g., the following variant of Example 4:

$$\{\texttt{A a} = [\texttt{new A()}]_4\texttt{;}\ \{\texttt{Pair p} = [\texttt{new Pair(a, a)}]_0\texttt{;}\ \texttt{new Pair(a, a)}\}\}|\emptyset$$

The local variable `p` is never used in the body of the block, so it makes sense for its initialization expression to be reduced with grade 0, since execution needs no copies of the result. Yet, the expression *needs to be reduced*, and to produce its useless result two copies of `a` are consumed; in a sense, they are wasted. However, such resource usage is tracked, whereas it would be lost if decrementing by 0. Removing the non-zero requirement would lead to a variant of resource-aware reduction where usage of resource which are useless to construct the final result is not tracked.

In rule (FIELD-ACCESS), the reduction grade should be (overapproximated by) the multiplication of the grade of the receiver with that of the field (constructor argument). Indeed, the former specifies how many copies of the object we have and the latter how many copies

$$
\begin{array}{lcll}
e & ::= & x \mid [e]_r.f \mid \texttt{new}\ C([e_1]_{r_1}, \ldots, [e_n]_{r_n}) \mid & \text{(annotated) expression} \\
 & & [e_0]_{r_0}.m([e_1]_{r_1}, \ldots, [e_n]_{r_n})\, es \mid \{T\, x = [e]_r;\ e'\} & \\
v & ::= & \texttt{new}\ C([v_1]_{r_1}, \ldots, [v_n]_{r_n}) & \text{(annotated) value}
\end{array}
$$

(VAR) $$\frac{}{x|\rho, x \mapsto \langle v, s\rangle \to_r v|\rho, x \mapsto \langle v, s'\rangle} \quad \begin{array}{l} r \preceq r' \neq \mathbf{0} \\ s' + r' \preceq s \end{array}$$

(FIELD-ACCESS) $$\frac{}{[\texttt{new}\ C([v_1]_{r_1}, \ldots, [v_n]_{r_n})]_r.f_i|\rho \to_s v_i|\rho} \quad \begin{array}{l} \mathsf{fields}(C) = T_1\ f_1; \ldots T_n\ f_n; \\ i \in 1..n \\ s \preceq r \cdot r_i \end{array}$$

(INVK) $$\frac{}{[v_0]_{r_0}.m([v_1]_{r_1}, \ldots, [v_n]_{r_n})|\rho \to_r e[y_0/\texttt{this}][y_1/x_1 \ldots y_n/x_n]|\rho'} \quad \begin{array}{l} v_0 = \texttt{new}\ C(\_) \\ \mathsf{mbody}(C, m) = \langle x_1 \ldots x_n, e\rangle \\ y_0, \ldots, y_n \notin \mathsf{dom}(\rho) \\ \rho' = \rho, y_0 \mapsto \langle v_0, r_0\rangle, \ldots, y_n \mapsto \langle v_n, r_n\rangle \end{array}$$

(BLOCK) $$\frac{}{\{C\, x = [v]_r;\ e\}|\rho \to_s e[y/x]|\rho, y \mapsto \langle v, r\rangle} \quad y \notin \mathsf{dom}(\rho)$$

(FIELD-ACCESS-CTX) $$\frac{e|\rho \to_r e'|\rho'}{[e]_r.f|\rho \to_s [e']_r.f|\rho'}$$

(NEW-CTX) $$\frac{e_i|\rho \to_{r \cdot r_i} e_i'|\rho'}{\begin{array}{l}\texttt{new}\ C([v_1]_{r_1}, \ldots, [v_{i-1}]_{r_{i-1}}, [e_i]_{r_i}, \ldots, [e_n]_{r_n})|\rho \to_r \\ \qquad \texttt{new}\ C([v_1]_{r_1}, \ldots, [v_{i-1}]_{r_{i-1}}, [e_i']_{r_i}, \ldots, [e_n]_{r_n})|\rho'\end{array}}$$

(INVK-RCV-CTX) $$\frac{e_0|\rho \to_{r_0} e_0'|\rho'}{[e_0]_{r_0}.m([e_1]_{r_1}, \ldots, [e_n]_{r_n})|\rho \to_r [e_0']_{r_0}.m([e_1]_{r_1}, \ldots, [e_n]_{r_n})|\rho'}$$

(INVK-ARG-CTX) $$\frac{e_i|\rho \to_{r_i} e_i'|\rho'}{\begin{array}{l}[e_0]_{r_0}.m([v_1]_{r_1}, \ldots, [v_{i-1}]_{r_{i-1}}, [e_i]_{r_i}, \ldots, [e_n]_{r_n})|\rho \to_r \\ \qquad [e_0]_{r_0}.m([v_1]_{r_1}, \ldots, [v_{i-1}]_{r_{i-1}}, [e_i']_{r_i}, \ldots, [e_n]_{r_n})|\rho'\end{array}}$$

(BLOCK-CTX) $$\frac{e_1|\rho \to_s e_1'|\rho'}{\{C\, x = [e_1]_s;\ e_2\}|\rho \to_r \{C\, x = [e_1']_s;\ e_2\}|\rho'}$$

Figure 2 Instrumented reduction.

of the field each of such objects has; thus, their product provides an upper bound to the grade of the resulting value. Note that, in this way, some reductions could be *forbidden*. For instance, taking the grade algebra of naturals, an access to a field whose value can be used 3 times, of an object reduced with grade 2, can be reduced with grade (at most) 6. Another more significant example is given in the following, taking the grade algebra of *privacy levels*.

Rule (INVK) adds each method parameter, including `this`, as available resource in the environment, modulo renaming with a fresh variable to avoid clashes. The associated value and grade are that of the corresponding argument. Rule (BLOCK) is exactly analogous, apart that only one variable is added.

Coming to contextual rules, the reduction grade of the subterm is that of the corresponding annotation, so that all steps happen with a fixed grade. The only exception is rule (NEW-CTX), where, symmetrically to rule (FIELD-ACCESS), the reduction grade for subterms should be the multiplication of the reduction grade of the object with the annotation of the field (constructor argument), capturing the intuition that the latter specifies the grade of the field for a single copy of the object. For instance, taking the grade algebra of naturals, to obtain an object which can be used twice, with a field which can be used 3 times, the value of such field should be an object which can be used 6 times.

Note that, besides the standard typing errors such as looking for a missing method or field, reduction graded r can get stuck since either rule (VAR) cannot be applied since the side conditions do not hold, or rule (FIELD-ACCESS) cannot be applied since the side condition $s \preceq r \cdot r_i$ does not hold. Informally, either some resource (variable) is exhausted, that is, can no longer be replaced by its value, or some field of some object cannot be extracted. It is also important to note that the instrumented reduction is non-deterministic, due to rule (VAR).

In the grade algebra used in the previous example, grades model *how many times* resources are used. However, grades can also model a non-quantitative[4] knowledge, that is, track possible *modes* in which a resource can be used, or, in other words, possible *constraints* on how it could be used. A typical example of this situation are *privacy levels*, which can be formalized similarly to what is done in [1], as described below.

▶ **Example 6.** Starting from any distributive semilattice lattice L, like in Example 2(5), define $\mathsf{L}_0 = \langle |\mathsf{L}_0|, \leq_0, \vee_0, \wedge_0, 0, \top \rangle$, where $|\mathsf{L}_0| = |\mathsf{L}| + \{0\}$ with $0 \leq_0 x$, $x \vee_0 0 = 0 \vee_0 x = x$ and $x \wedge_0 0 = 0 \wedge_0 x = 0$, for all $x \in |\mathsf{L}|$; on elements of $|\mathsf{L}|$ the order and the operations are those of L. That is, we assume that the privacy levels form a distributive semilattice with order representing "decreasing privacy", and we add a grade 0 modeling "non-used". The simplest instance consists of just two privacy levels, that is, $0 \preceq \mathsf{private} \preceq \mathsf{public}$. Sum is the join, meaning that we obtain a privacy level which is less restrictive than both: for instance, a variable which is used as public in a subterm, and as private in another, is overall used as public. Multiplication is the meet, meaning that we obtain a privacy level which is more restrictive than both: for instance, an access to a field whose value has been obtained in `public` mode, of an object reduced in `private` mode, is reduced in `private` mode[5]. Note that exactly the same structure could be used to model, e.g., rather than privacy levels, modifiers readonly and mutable in an imperative setting, corresponding to forbid field assignment and no restrictions, respectively. The following examples illustrates the use of such grade algebra. We write priv and pub for short, and classes `A` and `Pair` are as in the previous examples.

1. Let $e_1 = \{\mathtt{A\,y} = [\mathtt{new\ A()}]_{\mathsf{pub}}\mathtt{;}\ \{\mathtt{A\,x} = [\mathtt{y}]_{\mathsf{priv}}\mathtt{;}\ \mathtt{x}\}\}$ and `p_` be either pub or priv, e_1 starting with the empty environment reduces with grade private as follows:

$$\begin{array}{lll} e_1|\emptyset & \to_{\mathsf{priv}} & \{\mathtt{A\,x} = [\mathtt{y}]_{\mathsf{priv}}\mathtt{;}\ \mathtt{x}\}|\mathtt{y} \mapsto \langle \mathtt{new\ A()}, \mathsf{pub}\rangle \text{ with (BLOCK)} \\ & \to_{\mathsf{priv}} & \{\mathtt{A\,x} = [\mathtt{new\ A()}]_{\mathsf{priv}}\mathtt{;}\ \mathtt{x}\}|\mathtt{y} \mapsto \langle \mathtt{new\ A()}, \mathtt{p\_}\rangle \text{ with (BLOCK-CTX) and} \\ & & \mathtt{y}|\mathtt{y} \mapsto \langle \mathtt{new\ A()}, \mathsf{pub}\rangle \to_{\mathsf{priv}} \mathtt{new\ A()}|\mathtt{y} \mapsto \langle \mathtt{new\ A()}, \mathtt{p\_}\rangle \\ & \to_{\mathsf{priv}} & \mathtt{x}|\mathtt{y} \mapsto \langle \mathtt{new\ A()}, \mathtt{p\_}\rangle, \mathtt{x} \mapsto \langle \mathtt{new\ A()}, \mathsf{priv}\rangle \text{ with (BLOCK)} \\ & \to_{\mathsf{priv}} & \mathtt{new\ A()}|\mathtt{y} \mapsto \langle \mathtt{new\ A()}, \mathtt{p\_}\rangle, \mathtt{x} \mapsto \langle \mathtt{new\ A()}, \mathsf{priv}\rangle \text{ with (VAR)} \end{array}$$

   Instead reduction with grade public would be stuck since $\mathsf{pub} \not\preceq \mathsf{priv}$ and so

$$\mathtt{x}|\mathtt{y} \mapsto \langle \mathtt{new\ A()}, \mathtt{p\_}\rangle, \mathtt{x} \mapsto \langle \mathtt{new\ A()}, \mathsf{priv}\rangle \not\to_{\mathsf{pub}}$$

   Also the reduction of $e_2 = \{\mathtt{A\,y} = [\mathtt{new\ A()}]_{\mathsf{priv}}\mathtt{;}\ \{\mathtt{A\,x} = [\mathtt{y}]_{\mathsf{pub}}\mathtt{;}\ \mathtt{x}\}\}$ with grade private

$$\begin{array}{lll} e_2|\emptyset & \to_{\mathsf{priv}} & \{\mathtt{A\,x} = [\mathtt{y}]_{\mathsf{pub}}\mathtt{;}\ \mathtt{x}\}|\mathtt{y} \mapsto \langle \mathtt{new\ A()}, \mathsf{priv}\rangle \text{ with (BLOCK)} \\ & \not\to_{\mathsf{priv}} & \end{array}$$

   would be stuck since $\mathtt{y}|\mathtt{y} \mapsto \langle \mathtt{new\ A()}, \mathsf{priv}\rangle \not\to_{\mathsf{pub}}$. Note that both e_1 and e_2 reduce to `new A()` with the semantics of Figure 1.

[4] Suck kind of applications are called *informational* in [1].

[5] As in *viewpoint adaptation* [13], where permission to a field access can be restricted based on the permission to the base object.

2. Let $e_3 = \{\texttt{A x} = [\texttt{new A()}]_{\mathsf{pub}}\texttt{; new Pair(}[\texttt{x}]_{\mathsf{pub}}, [\texttt{x}]_{\mathsf{priv}}\texttt{)}\}$, e_3 starting with the empty environment reduces with grade public as follows:

$$\begin{aligned} e_3|\emptyset \quad &\to_{\mathsf{pub}} \quad \texttt{new Pair(}[\texttt{x}]_{\mathsf{pub}}, [\texttt{x}]_{\mathsf{priv}}\texttt{)}|\texttt{x} \mapsto \langle \texttt{new A()}, \mathsf{pub}\rangle \text{ with (Block)} \\ &\to_{\mathsf{pub}} \quad \texttt{new Pair(}[\texttt{new A()}]_{\mathsf{pub}}, [\texttt{x}]_{\mathsf{priv}}\texttt{)}|\texttt{x} \mapsto \langle \texttt{new A()}, \texttt{p\_}\rangle \text{ with (New-Ctx) and} \\ &\qquad\quad \texttt{x}|\texttt{x} \mapsto \langle \texttt{new A()}, \mathsf{pub}\rangle \to_{\mathsf{pub}} \texttt{new A()}|\texttt{x} \mapsto \langle \texttt{new A()}, \texttt{p\_}\rangle \\ &\to_{\mathsf{pub}} \quad \texttt{new Pair(}[\texttt{new A()}]_{\mathsf{pub}}, [\texttt{new A()}]_{\mathsf{priv}}\texttt{)}|\texttt{x} \mapsto \langle \texttt{new A()}, \texttt{p\_}\rangle \text{ with (New-Ctx) and} \\ &\qquad\quad \texttt{x}|\texttt{x} \mapsto \langle \texttt{new A()}, \texttt{p\_}\rangle \to_{\mathsf{priv}} \texttt{new A()}|\texttt{x} \mapsto \langle \texttt{new A()}, \texttt{p\_}\rangle \end{aligned}$$

It is easy to see that also $e_3|\emptyset \to^*_{\mathsf{priv}} \texttt{new Pair(}[\texttt{new A()}]_{\mathsf{pub}}, [\texttt{new A()}]_{\mathsf{priv}}\texttt{)}|\texttt{x} \mapsto \langle \texttt{new A()}, \texttt{p\_}\rangle$. So we have

$$[e_3]_r\texttt{.f}|\emptyset \quad \to^*_s \quad [\texttt{new Pair(}[\texttt{new A()}]_{\mathsf{pub}}, [\texttt{new A()}]_{\mathsf{priv}}\texttt{)}]_r\texttt{.f}|\texttt{x} \mapsto \langle \texttt{new A()}, \texttt{p\_}\rangle$$

where f can be either first or second and r and s can be either pub or priv. Now, the reductions of grade priv accessing either first or second produce the value of the fields

$$[\texttt{new Pair(}[\texttt{new A()}]_{\mathsf{pub}}, [\texttt{new A()}]_{\mathsf{priv}}\texttt{)}]_r\texttt{.f}|\_ \to_{\mathsf{priv}} \texttt{new A()}|\_$$

However, looking at the reductions of grade pub, only

$$[\texttt{new Pair(}[\texttt{new A()}]_{\mathsf{pub}}, [\texttt{new A()}]_{\mathsf{priv}}\texttt{)}]_{\mathsf{pub}}\texttt{.first}|\_ \to_{\mathsf{pub}} \texttt{new A()}|\_$$

is not stuck. That is, we produce a value that can be used as public only if we get a public field of a public object, whereas any value can be used as private.

We now state some simple properties of the semantics we will use to prove type soundness. The former establishes that reduction does not remove variables from the environment, the latter states that we can always decrease the grade of a reduction step.

▶ **Proposition 7.** *If $e|\rho \to_r e'|\rho'$ then $\mathit{dom}(\rho) \subseteq \mathit{dom}(\rho')$ and for all $x \in \mathit{dom}(\rho)$, $\rho(x) = \langle v, r\rangle$ implies $\rho'(x) = \langle v, s\rangle$ with $s \preceq r$.*

▶ **Proposition 8.** *If $e|\rho \to_r e'|\rho'$ and $s \preceq r$ then $e|\rho \to_s e'|\rho'$.*

We expect the instrumented reduction to be *sound* with respect to the standard reduction, in the sense that by erasing annotations from an instrumented reduction sequence we get a standard reduction sequence. This is formally stated below.

For any e expression, let us denote by $\lceil e \rceil$ the expression obtained by erasing annotations, defined in the obvious way, and analogously for environments, where grades associated to variables are removed as well.

▶ **Proposition 9** (Soundness of instrumented semantics)**.**
If $e|\rho \to_r e'|\rho'$, then $\lceil e \rceil|\lceil \rho \rceil \to \lceil e' \rceil|\lceil \rho' \rceil$.

The converse does not hold, since a configuration could be annotated in a way that makes it stuck; notably, some resource (variable) could be exhausted or some field of an object could not be extracted. The graded type system in the next section will generate annotations which ensure soundness, hence also completeness with respect to the standard reduction.

4 Graded Featherweight Java

Types (class names) are annotated with *grades*, as shown in Figure 3.

As anticipated at the end of Section 2, a *coeffect context*, of shape $\gamma = x_1 : r_1, \ldots, x_n : r_n$, where order is immaterial and $x_i \neq x_j$ for $i \neq j$, represents a map from variables to grades (called *coeffects* when used in this position) where only a finite number of variables have

$$
\begin{array}{lcll}
e & ::= & x \mid e.f \mid \texttt{new}\ C(es) \mid e.m(es) \mid \{T\,x = e;\ e'\} & \text{expression} \\
T & ::= & C^r & \text{(graded) type} \\
v & ::= & \texttt{new}\ C(vs) & \text{value}
\end{array}
$$

Figure 3 Syntax with grades.

non-zero coeffect. A *(type-and-coeffect) context*, of shape $\Gamma = x_1 :_{r_1} C_1, \ldots, x_n :_{r_n} C_n$, with analogous conventions, represents the pair of the standard type context $x_1 : C_1 \ldots, x_n : C_n$, and the coeffect context $x_1 : r_1, \ldots, x_n : r_n$. We write $\mathsf{dom}(\Gamma)$ for $\{x_1, \ldots, x_n\}$.

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:

- a partial order $\preceq$

$$
\begin{array}{rll}
\emptyset \preceq \emptyset & & \\
x :_s C, \Gamma \preceq x :_r C, \Delta & & \text{if } s \preceq r \text{ and } \Gamma \preceq \Delta \\
\Gamma \preceq x :_r C, \Delta & & \text{if } x \notin \mathsf{dom}(\Gamma) \text{ and } \Gamma \preceq \Delta
\end{array}
$$

- a sum $+$

$$
\begin{array}{rll}
\emptyset + \Gamma = \Gamma & & \\
(x :_s C, \Gamma) + (x :_r C, \Delta) = x :_{s+r} C, (\Gamma + \Delta) & & \\
(x :_s C, \Gamma) + \Delta = x :_s C, (\Gamma + \Delta) & & \text{if } x \notin \mathsf{dom}(\Delta)
\end{array}
$$

- a scalar multiplication $\cdot$

$$
s \cdot \emptyset = \emptyset \qquad\qquad s \cdot (x :_r C, \Gamma) = x :_{s \cdot r} C, (s \cdot \Gamma)
$$

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 addition becomes partial since a variable in the domain of both contexts is required to have the same type.

The type system relies on the type information extracted from the class table, which, again to be concise, is abstractly modeled as follows:

- the subtyping relation $\leq$ on class names is the reflexive and transitive closure of the `extends` relation
- $\mathsf{mtype}(C, m)$ gives, for each method m of class C, its enriched method type, where the types of the parameters and of `this` have coeffect annotations.

Moreover, $\mathsf{fields}(C)$ gives now a sequence $C_1^{r_1}\ f_1\texttt{;} \ldots C_n^{r_n}\ f_n\texttt{;}$, meaning that, to construct an object of type C, we need to provide, for each $i \in 1..n$, a value with a grade at least r_i.

The subtyping relation on graded types is defined as follows:

$$C^r \leq D^s \text{ iff } C \leq D \text{ and } s \preceq r$$

That is, a graded type is a subtype of another if the class is a heir class and the grade is more constraining. For instance, taking the affinity grade algebra of Example 2(2), an invocation of a method with return type C^{ω} can be used in a context where a type $C^{\mathbf{1}}$ is required, e.g., to initialize a $C^{\mathbf{1}}$ variable.

The typing judgment has shape $\Gamma \vdash e : T \rightsquigarrow e'$, where Γ is a type-and-coeffect context, and e' is an annotated expression, as defined in Figure 2. That is, typechecking generates annotations in code such that evaluation cannot get stuck, as will be formally expressed and proved in the following.

$$\text{(T-SUB)}\ \frac{\Gamma \vdash e : T \rightsquigarrow e' \quad \Gamma \preceq \Gamma' \quad T \leq T'}{\Gamma' \vdash e : T' \rightsquigarrow e'} \qquad \text{(T-VAR)}\ \frac{}{x :_r C \vdash x : C^r \rightsquigarrow x}\ r \neq \mathbf{0}$$

$$\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash e : C^r \rightsquigarrow e'}{\Gamma \vdash e.f_i : C_i^{r \cdot r_i} \rightsquigarrow [e']_r.f_i}\ \mathsf{fields}(C) = C_1^{r_1}\ f_1; \dots C_n^{r_n}\ f_n;$$

$$\text{(T-NEW)}\ \frac{\Gamma_i \vdash e_i : C_i^{r \cdot r_i} \rightsquigarrow e_i' \quad \forall i \in 1..n}{\Gamma_1 + \ldots + \Gamma_n \vdash \mathtt{new}\ C(e_1, \ldots, e_n) : C^r \rightsquigarrow \mathtt{new}\ C([e_1']_{r_1}, \ldots, [e_n']_{r_n})}\ \mathsf{fields}(C) = C_1^{r_1}\ f_1; \dots C_n^{r_n}\ f_n;$$

$$\text{(T-INVK)}\ \frac{\Gamma_0 \vdash e_0 : C^{r_0} \rightsquigarrow e_0' \quad \Gamma_i \vdash e_i : C_i^{r_i} \rightsquigarrow e_i' \quad \forall 1 \in 1..n}{\Gamma_0 + \ldots + \Gamma_n \vdash e_0.m(e_1, \ldots, e_n) : T \rightsquigarrow [e_0']_{r_0}.m([e_1]_{r_1}, \ldots, [e_n]_{r_n})}\ \mathsf{mtype}(C, m) = r_0, C_1^{r_1} \dots C_n^{r_n} \to T$$

$$\text{(T-BLOCK)}\ \frac{\Gamma_1 \vdash e_1 : C^r \rightsquigarrow e_1' \quad \Gamma_2, x :_r C \vdash e_2 : T \rightsquigarrow e_2'}{\Gamma_1 + \Gamma_2 \vdash \{C^r\ x = e_1; e_2\} : T \rightsquigarrow \{C\ x = [e_1']_r; e_2'\}}$$

$$\text{(T-ENV)}\ \frac{\vdash v_i : C_i^{r_i} \rightsquigarrow v_i' \quad \forall i \in 1..n}{\Gamma \vdash \rho \rightsquigarrow \rho'}\ \begin{array}{l}\Gamma = x_1 :_{r_1} C_1, \ldots, x_n :_{r_n} C_n \\ \rho = x_1 \mapsto \langle v_1, r_1\rangle, \ldots, x_n \mapsto \langle v_n, r_n\rangle \\ \rho' = x_1 \mapsto \langle v_1', r_1\rangle, \ldots, x_n \mapsto \langle v_n', r_n\rangle\end{array}$$

$$\text{(T-CONF)}\ \frac{\Delta \vdash e : T \rightsquigarrow e' \quad \Gamma \vdash \rho \rightsquigarrow \rho'}{\Gamma \vdash e|\rho : T \rightsquigarrow e'|\rho'}\ \Delta \preceq \Gamma$$

Figure 4 Graded type system.

In a well-typed class table, method bodies are expected to conform to method types. That is, $\mathsf{mtype}(C, m)$ 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 $\mathsf{mbody}(C, m) = \langle x_1 \dots x_n, e\rangle$, and $\mathsf{mtype}(C, m) = r_0, C_1^{r_1} \dots C_n^{r_n} \to T$, then the following condition must hold:

$$\text{(T-METH)}\ \ \mathtt{this} :_{r_0} C, x_1 :_{r_1} C_1, \ldots, x_n :_{r_n} C_n \vdash e : T \rightsquigarrow e'$$

Moreover, we assume the standard coherence conditions on the class table with respect to inheritance. That is, if $C \leq D$, then $\mathsf{fields}(D)$ is a prefix of $\mathsf{fields}(C)$ and, if $\mathsf{mtype}(C, m) = r_0, C_1^{r_1} \dots C_n^{r_n} \to T$, then $\mathsf{mtype}(D, m) = r_0, C_1^{r_1} \dots C_n^{r_n} \to T'$ with $T' \leq T$.

In Figure 4, we describe the typing rules, which are *parameterized* on the underlying grade algebra.

In rule (T-SUB), both the coeffect context and the (graded) type can be made more general. This means that, on one hand, 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 as well with no constraints (coeffect ω). On the other hand, recalling that grades are contravariant in types, an expression can get a more constraining grade. For instance, an expression of grade ω can be used where a grade 1 is required.

If we take $r = \mathbf{1}$, then rule (T-VAR) is analogous to the standard rule for variable in coeffect systems, where the coeffect context is the map where the given variable is used once, and no other is used. Here, more generally, the variable can get an arbitrary grade r, provided that it gets the same grade in the context. However, the use of the variable cannot be just discarded, as expressed by the side condition $r \neq \mathbf{0}$.

In rule (T-FIELD-ACCESS), the grade of the field is multiplied by the grade of the receiver. As already mentioned, this is a form of *viewpoint adaptation* [13]. For instance, using affinity grades, a field graded ω of an object graded 1 can be used at most once.

In rule (T-NEW), analogously to rule (T-VAR), the constructor invocation can get an arbitrary grade r, provided that the grades of the fields are multiplied by the same grade. Coeffects of the subterms are summed, as customary in type-and-coeffect systems.

In rule (T-INVK), the coeffects of the arguments are summed as well. The rule uses the function mtype on the class table, which, given a class name and a method name, returns its parameter and return (graded) types. For the implicit parameter `this` only the grade is specified. Note that the grades of the parameters are used in two different ways: as (part of) types, when typechecking the arguments; as coeffects, when typechecking the method body.

In rule (T-BLOCK), the coeffects of the initialization expression are summed with those of the body, excluding the local variable. Analogously to method parameters, the grade of the local variable is both used as (part of) type, when typechecking the initialization expression, and as coeffect, when typechecking the body.

Finally, we have straightforward rules for typing environments and configurations. Values in the environment are assumed to be closed, since we are in a call-by-value calculus. Also note that, in the judgment for environments and configurations, since no subsumption rule is available, variables in the context are exactly those in the domain of the environment, which are a superset of those used in the expression.

▶ **Example 10.** We show a simple example illustrating the use of graded types, assuming affinity grades. We write in square brackets the grade of the implicit `this` parameter. The class `Pair` declares three versions of the getter for the `first` field, which differ for the grade of the result: either 0, meaning that the result of the method *cannot* be used, or 1, meaning it can be used at most once, or ω, meaning it can be used with no constraints. Note that the first version, clearly useless in a functional calculus, could make sense adding effects, e.g. in an imperative calculus, playing a role similar to that of `void`.

```
class Pair { A^1 first; A^1 second;
  A^0 getFirstZero() [1]{this.first}
  A^1 getFirstAffine() [1]{this.first}
  A^ω getFirst() [1]{this.first}
}
```

The coeffect of `this` is 1 in all versions, and it is actually used once in the bodies. The occurrence of `this` in the bodies can get any non-zero grade thanks to rule (T-VAR), and fields are graded 1, meaning that a field access does not affect the grade of the receiver, hence the three bodies can get any non-zero grade as well, so they are well-typed with respect to the grade in the method return type.

In the client code below, a call of the getter is assigned to a local variable of the same grade, which is then used consistently with such grade.

```
Pair^1 p = ...

{A^0 a = p.getFirstZero(); new Pair(new A(),new A())}
{A^1 a = p.getFirstAffine(); new Pair(a,new A())}
{A^ω a = p.getFirst(); new Pair(a,a)}
```

The following blocks are, instead, ill-typed, for two different reasons.

```
{A^1 a = p.getFirst(); new Pair(a,a)}
{A^ω a = p.getFirstAffine(); new Pair(a,a)}
```

In the first one, the initialization is correct, by subsumption, since we use an expression of a less constrained grade. However, the variable is then used in a way which is not compatible with its grade. In the second one, instead, the variable is used consistently with its grade, but the initialization is ill-typed, since we use an expression of a more constrained grade.

Finally, note that the coeffect of `this` could be safely changed to be ω in the three methods, providing an overapproximated information; in this case, however, the three invocations in the client code would be wrong, since the receiver `p` is required to be used at most once.

▶ **Example 11.** Consider the following source (that is, non-annotated) version of the expression in Example 5.

```
{A^public y = new A(); {A^private x = y; x}}
```

The private variable `x` is initialized with the public expression/variable `y`. The block expression has type $\mathtt{A}^{\mathsf{private}}$ as the following type derivation shows.

$$\text{(T-BLOCK)}\ \frac{\text{(T-NEW)}\ \dfrac{}{\vdash \mathtt{new\ A()} : \mathtt{A}^{\mathsf{pub}}} \qquad \mathcal{D}}{\vdash \{\mathtt{A}^{\mathsf{pub}}\ \mathtt{y} = \mathtt{new\ A()};\ \{\mathtt{A}^{\mathsf{priv}}\ \mathtt{x} = \mathtt{y};\ \mathtt{x}\}\} : \mathtt{A}^{\mathsf{priv}}}$$

where $\mathcal{D}$ is the following derivation

$$\text{(T-BLOCK)}\ \frac{\text{(T-SUB)}\ \dfrac{\text{(T-VAR)}\ \dfrac{}{\mathtt{y} :_{\mathsf{pub}} \mathtt{A} \vdash \mathtt{y} : \mathtt{A}^{\mathsf{pub}}}}{\mathtt{y} :_{\mathsf{pub}} \mathtt{A} \vdash \mathtt{y} : \mathtt{A}^{\mathsf{priv}}} \qquad \text{(T-VAR)}\ \dfrac{}{\mathtt{y} :_{\mathsf{pub}} \mathtt{A}, \mathtt{x} :_{\mathsf{priv}} \mathtt{A} \vdash \mathtt{x} : \mathtt{A}^{\mathsf{priv}}}}{\mathtt{y} :_{\mathsf{pub}} \mathtt{A} \vdash \{\mathtt{A}^{\mathsf{priv}}\ \mathtt{x} = \mathtt{y};\ \mathtt{x}\} : \mathtt{A}^{\mathsf{priv}}}$$

On the other hand, initializing a public variable with a private expression as in

```
{A^private y = new A(); {A^public x = y; x}}
```

is not possible, as expected, since $\mathtt{y} :_{\mathsf{priv}} \mathtt{A} \not\vdash \mathtt{y} : \mathtt{A}^{\mathsf{pub}}$.

Consider now the class `Pair` with a private field and a public one.

```
class B { A^public f1; A^private f2; }
```

The expression e

```
{A^public x = new A(); new B(x,x)}
```

can be given type $\mathtt{Pair}^{\mathsf{public}}$ as follows:

$$\text{(T-BLOCK)}\ \frac{\text{(T-NEW)}\ \dfrac{}{\vdash \mathtt{new\ A()} : \mathtt{A}^{\mathsf{pub}}} \quad \text{(T-NEW)}\ \dfrac{\text{(T-VAR)}\ \dfrac{}{\mathtt{x} :_{\mathsf{pub}} \mathtt{A} \vdash \mathtt{x} : \mathtt{A}^{\mathsf{pub}}} \quad \text{(T-SUB)}\ \dfrac{\text{(T-VAR)}\ \dfrac{}{\mathtt{x} :_{\mathsf{pub}} \mathtt{A} \vdash \mathtt{x} : \mathtt{A}^{\mathsf{pub}}}}{\mathtt{x} :_{\mathsf{pub}} \mathtt{A} \vdash \mathtt{x} : \mathtt{A}^{\mathsf{priv}}}}{\mathtt{x} :_{\mathsf{pub}} \mathtt{A} \vdash \mathtt{new\ Pair(x,x)} : \mathtt{Pair}^{\mathsf{pub}}}}{\vdash \{\mathtt{A}^{\mathsf{pub}}\ \mathtt{x} = \mathtt{new\ A()};\ \mathtt{new\ Pair(x,x)}\} : \mathtt{Pair}^{\mathsf{pub}}}$$

By (T-SUB) we can also derive $\vdash e : \mathtt{Pair}^{\mathsf{priv}}$ and so we get

$$\text{(T-FIELD)}\ \frac{\vdash e : \mathtt{Pair}^{\mathsf{priv}}}{\vdash e.\mathtt{first} : \mathtt{A}^{\mathsf{priv}}} \qquad \text{(T-FIELD)}\ \frac{\vdash e : \mathtt{Pair}^{\mathsf{pub}}}{\vdash e.\mathtt{second} : \mathtt{A}^{\mathsf{priv}}}$$

that is, accessing a public field of a private expression we get a private result as well as accessing a private field of a public expression.
Also note that the following expression e'

```
{A^private x = new A(); new B(x,x)}
```

can be given only type $\mathtt{Pair}^{\mathsf{private}}$ by

$$\text{(T-BLOCK)}\ \frac{\text{(T-NEW)}\ \dfrac{}{\vdash \mathtt{new\ A()} : \mathtt{A}^{\mathsf{priv}}} \quad \text{(T-NEW)}\ \dfrac{\text{(T-VAR)}\ \dfrac{}{\mathtt{x} :_{\mathsf{priv}} \mathtt{A} \vdash \mathtt{x} : \mathtt{A}^{\mathsf{priv}}} \quad \text{(T-VAR)}\ \dfrac{}{\mathtt{x} :_{\mathsf{priv}} \mathtt{A} \vdash \mathtt{x} : \mathtt{A}^{\mathsf{priv}}}}{\mathtt{x} :_{\mathsf{priv}} \mathtt{A} \vdash \mathtt{new\ Pair(x,x)} : \mathtt{Pair}^{\mathsf{priv}}}}{\vdash \{\mathtt{A}^{\mathsf{priv}}\ \mathtt{x} = \mathtt{new\ A()};\ \mathtt{new\ Pair(x,x)}\} : \mathtt{Pair}^{\mathsf{priv}}}$$

We cannot derive $\vdash e' : \mathtt{Pair}^{\mathsf{pub}}$, since the grade of `first` is public and (T-NEW) would require $\mathtt{x} :_{\mathsf{priv}} \mathtt{A} \vdash \mathtt{x} : \mathtt{A}^{\mathsf{pub}\cdot\mathsf{pub}}$, which does not hold.

$$\text{(T-SUB)}\ \frac{\Gamma \vdash_a e : T \quad \Gamma \preceq \Gamma'}{\Gamma' \vdash_a e : T' \quad T \leq T'} \qquad \text{(T-VAR)}\ \frac{}{x :_r C \vdash_a x : C^r}\ r \neq \mathbf{0}$$

$$\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash_a e : C^r}{\Gamma \vdash_a [e]_r.f_i : C_i^{r \cdot r_i}}\ \mathsf{fields}(C) = C_1^{r_1}\ f_1; \ldots C_n^{r_n}\ f_n;$$

$$\text{(T-NEW)}\ \frac{\Gamma_i \vdash_a e_i : C_i^{r \cdot r_i} \quad \forall i \in 1..n}{\Gamma_1 + \ldots + \Gamma_n \vdash_a \texttt{new}\ C([e_1]_{r_1}, \ldots, [e_n]_{r_n}) : C^r}\ \mathsf{fields}(C) = C_1^{r_1}\ f_1; \ldots C_n^{r_n}\ f_n;$$

$$\text{(T-INVK)}\ \frac{\Gamma_0 \vdash_a e_0 : C^{r_0} \quad \Gamma_i \vdash_a e_i : C_i^{r_i} \quad \forall 1 \in 1..n}{\Gamma_0 + \ldots + \Gamma_n \vdash_a [e_0]_{r_0}.m([e_1]_{r_1}, \ldots, [e_n]_{r_n}) : T}\ \mathsf{mtype}(C, m) = r_0, C_1^{r_1} \ldots C_n^{r_n} \to T$$

$$\text{(T-BLOCK)}\ \frac{\Gamma_1 \vdash_a e_1 : C^r \quad \Gamma_2, x :_r C \vdash_a e_2 : T}{\Gamma_1 + \Gamma_2 \vdash_a \{C\, x = [e_1]_r;\ e_2\} : T}$$

$$\text{(T-ENV)}\ \frac{\vdash_a v_i : C_i^{r_i} \quad \forall i \in 1..n}{\Gamma \vdash_a \rho}\ \begin{array}{l}\Gamma = x_1 :_{r_1} C_1, \ldots, x_n :_{r_n} C_n \\ \rho = x_1 \mapsto \langle v_1, r_1 \rangle, \ldots, x_n \mapsto \langle v_n, r_n \rangle\end{array}$$

$$\text{(T-CONF)}\ \frac{\Delta \vdash_a e : T \quad \Gamma \vdash_a \rho}{\Gamma \vdash_a e|\rho : T}\ \Delta \preceq \Gamma$$

Figure 5 Graded type system for annotated syntax.

Resource-aware soundness. We state that the graded type system is sound with respect to the resource-aware semantics. In other words, the graded type system prevents both standard typing errors, such as invoking a missing field or method, and resource-usage errors, such as requiring a resource which is exhausted (cannot be used in the needed way).

In order to state and prove a soundness theorem, we need to introduce a (straightforward) typing judgment $\vdash_a$ for annotated expressions, environments and configurations. The typing rules are reported in Figure 5.

Recall that $\lceil \_ \rceil$ denotes erasing annotations. It is easy to see that an annotated expression is well-typed if and only if it is produced by the type system:

▶ **Proposition 12.** *$\Gamma \vdash e : T \rightsquigarrow e'$ if and only if $\lceil e' \rceil = e$ and $\Gamma \vdash_a e' : T$.*

A similar property holds for environments and configurations.

The main result is the following resource-aware progress theorem.

▶ **Theorem 13** (Resource-aware progress)**.** *If $\Gamma \vdash_a e|\rho : C^r$ then either e is a value or $e|\rho \to_r e'|\rho'$ and $\Gamma' \vdash_a e'|\rho' : C^r$ with $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Gamma')$ and $\Gamma' \preceq \Gamma, \Delta$.*

When reduction is non-deterministic, we can distinguish two flavours of soundness, *soundness-must* meaning that no computation can be stuck, and *soundness-may*, meaning that at least one computation is not stuck. The terminology of *may* and *must* properties is very general and comes originally from [12]; the specific names soundness-may and soundness-must were introduced in [10, 9] in the context of big-step semantics. In our case, graded reduction is non-deterministic since, as discussed before, the rule (VAR) could be instantiated in different ways, possibly consuming the resource more than necessary. However, we expect that, for a well-typed configuration, there is at least one computation which is not stuck, hence a soundness-may result. Soundness-may can be proved by a theorem like the one above, which can be seen as a subject-reduction-may result, including standard progress. In our case, if the configuration is well-typed, that is, annotations have been generated by the type system, *there is a step* which leads, in turn, to a well-typed configuration. More in detail, the type is preserved, resources initially available may have reduced grades, and other available resources may be added.

Theorem 13 is proved as a special case of the following more general result, which makes explicit the invariant needed to carry out the induction. Indeed, by looking at the reduction rules, we can see that computational ones either add new variables to the environment or reduce the grade of a variable of some amount that depends on the grade of the reduction. In the latter case, the amount can be arbitrarily chosen with the only restrictions that it is non zero and at least the grade of the reduction. However, to prove progress, we not only have to prove that a reduction can be done, but, if the reduction is done in a context, say evaluating the argument of a constructor, then after such reduction we still have enough resources to go on with the reduction, that is, to evaluate the rest of the context (the other arguments of the constructor). This means that the resulting environment has enough resources to type the whole context (the constructor call). For this reason, in the statement of the theorem that follows, we add to the assumption of Theorem 13 a typing context Θ that would contain the information on the amount of resources that we want to preserve during the reduction (see Item 4 of the theorem). This allows us to choose the appropriate grade to be kept when reducing a variable and to reconstruct a typing derivation when using contextual reduction rules. For the expression at the top level, as we see from the proof of Theorem 13, Θ is simply $\mathbf{0}$ for all variables in the typing context in which the expression is typed.

▶ **Theorem 14.** *If* $\Delta \vdash_a e : C^r$ *and* $\Gamma \vdash_a \rho$ *and* $\Delta + \Theta \preceq \Gamma$ *and* $\mathit{dom}(\Delta) \subseteq \mathit{dom}(\Theta)$ *and* e *is not a value, then there are* e', ρ', Δ', Γ' *and* Θ' *such that*

1. $e|\rho \to_r e'|\rho'$ *and*
2. $\Delta' \vdash_a e' : C^r$ *with* $\Delta' \preceq \Delta, \Theta'$ *and*
3. $\Gamma' \vdash_a \rho'$ *with* $\Gamma' \preceq \Gamma, \Theta'$ *and*
4. $\Delta' + \Theta \preceq \Gamma'$.

Finally, the following corollary states both subject-reduction for the standard semantics, that is, type and coeffects are preserved, and completeness of the instrumented semantics, that is, for well-typed configurations, every reduction step in the usual semantics can be simulated by an appropriate step in the instrumented semantics.

▶ **Corollary 15** (Subject reduction). *If* $\Gamma_1 \vdash e_1|\rho_1 : C^r \rightsquigarrow e_1'|\rho_1'$ *and* $e_1|\rho_1 \to e_2|\rho_2$, *then* $\Gamma_2 \vdash e_2|\rho_2 : C^r \rightsquigarrow e_2'|\rho_2'$ *with* $\mathit{dom}(\Gamma_1) \subseteq \mathit{dom}(\Gamma_2)$ *and* $\Gamma_2 \preceq \Gamma_1, \Delta$, *and* $e_1'|\rho_1' \to_r e_2'|\rho_2'$.

Proof. By Proposition 12 we get $\Gamma_1 \vdash_a e_1'|\rho_1' : C^r$ and, by Theorem 13, $e_1'|\rho_1' \to_r e_2'|\rho_2'$ and $\Gamma_2 \vdash_a e_2'|\rho_2' : C^r$ with $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(\Gamma_2)$ and $\Gamma_2 \preceq \Gamma_1, \Delta$. By Proposition 12, we get $\Gamma_2 \vdash \lceil e_2' \rceil | \lceil \rho_2' \rceil : C^r \rightsquigarrow e_2'|\rho_2'$ and by Proposition 9, we get $e_1|\rho_1 \to \lceil e_2' \rceil | \lceil \rho_2' \rceil$. By the determinism of the standard semantics we have $\lceil e_2' \rceil = e_2$ and $\lceil \rho_2' \rceil = \rho_2$, hence the thesis. ◀

5 Combining grades

As we have seen, each grade algebra encodes a specific notion of resource usage. However, a program may need different notions of usage for different resources or different pieces of code (e.g., different classes). Hence, one needs to use several grade algebras at the same time, that is, a family $(H_k)_{k \in \mathcal{K}}$ of grade algebras[6] indexed over a set $\mathcal{K}$ of *grade kinds*. We assume grade kinds to always include N and T, with H_{N} and H_{T} the grade algebras of natural numbers and trivial, respectively, as in Example 2, since they play a special role, as will be shown.

[6] H stands for "heterogeneous".

▶ **Example 16.** Assume to use, in a program, grade kinds N, A, P, PP, AP, and T, where:

- H_{A} is the affinity grade algebra, as in Example 2(3).
- H_{P} and H_{PP} are two different instantiations of the grade algebra of privacy levels, as in Example 6; namely, in H_{P} there are only two privacy levels public and $\mathsf{private}$, whereas in H_{PP} we have privacy levels a, b, c, d, with $\mathsf{a} \preceq \mathsf{b} \preceq \mathsf{d}$ and $\mathsf{a} \preceq \mathsf{c} \preceq \mathsf{d}$.
- Finally, H_{AP} is $H_{\mathsf{A}} \times H_{\mathsf{P}}$, as in Example 2(7), tracking simultaneously affinity and privacy.

We want to make grades of all such kinds simultaneously available to the programmer. In order to achieve this, we should specify how to *combine* grades of different kinds through their distinctive operators; for instance, an object with grade of kind k could have a field with grade of kind μ, hence a field access should be graded by their multiplication.

In other words, we need to construct, starting from the family $(H_k)_{k \in K}$, a single grade algebra of *heterogeneous grades*. In this way, the meta-theory developed in previous sections for an arbitrary grade algebra applies also to the case when several grade algebras are used at the same time. Note that this construction is necessary since we do not want available grades to be fixed, as in [23]; rather, the programmer should be allowed to define grades for a specific application, using some linguistic support which could be the language itself, as will be described in Section 6.

Direct refinement. The obvious approach is to define heterogeneous grades as pairs $\langle k, r \rangle$ where $k \in \mathcal{K}$, and $r \in H_k$. Concerning operators, in previous work, handling coeffects rather than grades, [3] we took the simplest choice, that is, combining (by either sum or product) grades of different kinds always returns $\langle \infty, \mathsf{T} \rangle$, meaning, in a sense, that we "do not know" how the combination should be done. The only exception are grades of kind N; indeed, since the corresponding grade algebra is initial, we know that, for any kind k, there is a unique grade homomorphism ι_k from Nat to H_k, hence, to combine $\langle n, \mathsf{N} \rangle$ with $\langle r, k \rangle$, we can map n into a grade of kind k through such homomorphism, and then use the operator of kind k. In this paper, we generalize this idea, by allowing the programmer to specify, for each pair of kinds k and μ, a uniquely determined kind $k \oplus \mu$ and two uniquely determined grade homomorphisms $\mathsf{lh}^H_{\kappa,\mu} \colon H_\kappa \to H_{\kappa \oplus \mu}$, and $\mathsf{rh}^H_{\kappa,\mu} \colon H_\mu \to H_{\kappa \oplus \mu}$. In this way, to combine $\langle \kappa, r \rangle$ and $\langle \mu, s \rangle$, we can map both in grades of kind $k \oplus \mu$, and then use the operator of kind $k \oplus \mu$.

The operator $\oplus$ and the family of unique homomorphisms, one for each pair of kinds, can be specified by the programmer, in a minimal and easy to check way, by defining a *(direct) refinement relation* $\sqsubset^1$, as defined below, and a family of grade homomorphisms $H_{\kappa,\mu} \colon H_\kappa \to H_\mu$, indexed over pairs $\kappa \sqsubset^1 \mu$.

Given a relation $\Rightarrow$ on kinds, a *path* from k_0 to k_n is a sequence $k_0 \dots k_n$ such as $k_i \Rightarrow k_{i+1}$, for all $i \in 1..n-1$. We say that μ is an *ancestor* of κ if there is a path from κ to μ.

Then, a *(direct) refinement relation* is a relation $\sqsubset^1$ on $\mathcal{K} \setminus \{\mathsf{N}, \mathsf{T}\}$ such as the following conditions hold:

1. for each κ, μ, there exists at most one path from κ to μ
2. for each κ, μ with a common ancestor, there is a *least* common ancestor, denoted $\kappa \oplus \mu$; that is, such that, for any common ancestor ν, ν is an ancestor of $k \oplus \mu$.

Note that, thanks to requirement (1), requirement (2) means that the unique path, e.g., from κ to ν, consists of a unique path from κ to $k \oplus \mu$, and then a unique path from $k \oplus \mu$ to ν.

Given a direct refinement relation $\sqsubset^1$, we can derive the following structure on $\mathcal{K}$:

- $\sqsubset^1$ can be extended to a partial order $\sqsubseteq$ on $\mathcal{K}$, by taking the reflexive and transitive closure of $\sqsubset^1$ and adding $\mathsf{N} \sqsubseteq \kappa \sqsubseteq \mathsf{T}$ for all $\kappa \in \mathcal{K}$.
- $\oplus$ can be extended to all pairs, by defining $\kappa \oplus \mu = \mathsf{T}$ if κ and μ have no common ancestor.

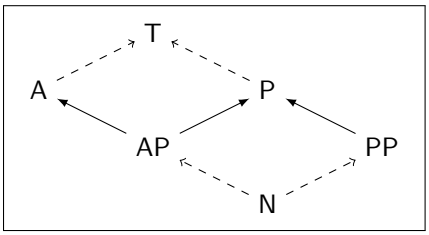

Figure 6 Direct refinement diagram.

Altogether, we obtain an instance of a structure called *grade signature*, as will be detailed in Definition 18. Moreover, given a $\sqsubset^1$-family of homomorphisms:

- they can be extended, by composition[7], to all pairs of grades $\langle \kappa, \mu \rangle \in \mathcal{K} \setminus \{\mathsf{N}, \mathsf{T}\}$ such that there is a path from κ to μ; since this path is unique, the resulting homomorphism is uniquely defined
- for each kind κ, we add the unique homomorphisms from Nat and to Triv.

Altogether, besides a grade algebra for each kind, we get a grade homomophism for each pair $\langle \kappa, \mu \rangle$ such that $\kappa \sqsubseteq \mu$. That is, we obtain an instance of a structure called *heterogeneous grade algebra*, as will be detailed in Definition 19.

Thus, as desired, combining grades of kinds $\langle \kappa, r \rangle$ and $\langle \mu, s \rangle$ can be defined by mapping both r and s into grades of kind $\kappa \oplus \mu$, and then the operator of kind $\kappa \oplus \mu$ is applied.

The fact that in this way we actually obtain a grade algebra, that is, all required axioms are satisfied, is proved in the next subsection on the more general case of an arbitrary grade signature and heterogeneous grade algebra.

Note the special role played by the grade kinds N and T, with their corresponding grade algebras. The former turns out to be the minimal kind required in a grade signature (Definition 18); this is important since the zero and one of the resulting grade algebra (hence the zero and one used in the type system) will be those of this kind. The latter, as shown above, is used as default common ancestor for pairs of kinds which do not have one.

▶ **Example 17.** Coming back to our example, a programmer could define the direct refinement relation and the corresponding homomorphisms as follows:

- $\mathsf{PP} \sqsubset^1 \mathsf{P}$, and the homomorphism maps, e.g., a, b, and c into $\mathsf{private}$ and d into public
- $\mathsf{AP} \sqsubset^1 \mathsf{A}$, and $\mathsf{AP} \sqsubset^1 \mathsf{P}$, and the homomorphisms are the projections.

Thus, for instance, the grade $\langle \mathsf{AP}, \langle \omega, \mathsf{private} \rangle \rangle$, meaning that we can use the resource an arbitrary number of times in $\mathsf{private}$ mode, and $\langle \mathsf{PP}, \mathsf{d} \rangle$, meaning that we can use the resource in d mode, gives $\mathsf{private}$. Indeed, both grades are mapped into the grade algebra of privacy levels $0 \preceq \mathsf{private} \preceq \mathsf{public}$; for the former, the information about the affinity is lost, whereas for the second the privacy level d is mapped into public; finally, we get $\mathsf{private} = \mathsf{private} \cdot \mathsf{public}$.

The direct refinement relation is pictorially shown in Figure 6. Dotted arrows denote (some of) the order relations added for N and T.

Note that specifying the grade signature and the heterogeneous grade algebra indirectly, by means of the direct refinement relation and the corresponding homomorphisms, has a fundamental advantage: the semantic check that, for each κ, μ, we can map grades of grade κ into grades of kind μ in a unique way (that is, there is at most one homomorphism), which would require checking the equivalence of function definitions, is replaced by the checks (1)

[7] Note that in this way we obtain, in particular, all the identities.

and (2) in the definition of direct refinement, which are purely syntactic and can be easily implemented in a type system (a simple stronger condition is to impose that each kind has a unique parent in the direct refinement relation, as it is for single inheritance).

In Section 6, we will see how to express both grade algebras and homomorphisms in Java; roughly, both will be represented by classes implementing a suitable generic interface.

A general construction. We provide a construction that, starting from a family of grade algebras with a suitable structure, yields a unique grade algebra summarising the whole family. As a consequence, the meta-theory developed in previous sections for a single grade algebra applies also to the case when several grade algebras are used at the same time.

To develop this construction, we use simple and standard categorical tools, referring to [20, 26] for more details. Given a category $\mathcal{C}$, we denote by $\mathcal{C}_0$ the collection of objects in $\mathcal{C}$ and we say that $\mathcal{C}$ is *small* when $\mathcal{C}_0$ is a set. Recall that any partially ordered set $\mathcal{P} = \langle \mathcal{P}_0, \sqsubseteq \rangle$ can be seen as a small category where objects are the elements of $\mathcal{P}_0$ and, for all $x, y \in \mathcal{P}_0$, there is an arrow $x \to y$ iff $x \sqsubseteq y$; hence, for every pair of objects in $\mathcal{P}_0$, there is at most one arrow between them, and the only isomorphisms are the identities.

▶ **Definition 18.** *A* grade signature $\mathcal{S}$ *is a partially ordered set with finite suprema, that is, it consists of the following data:*

- *a partially ordered set* $\langle \mathcal{S}_0, \sqsubseteq \rangle$*;*
- *a function* $\oplus \colon \mathcal{S}_0 \times \mathcal{S}_0 \to \mathcal{S}_0$ *monotone in both arguments and such that for all* $\kappa, \mu, \nu \in \mathcal{S}_0$*,* $\kappa \oplus \mu \sqsubseteq \nu$ *iff* $\kappa \sqsubseteq \nu$ *and* $\mu \sqsubseteq \nu$*;*
- *a distinguished object* $I \in \mathcal{S}_0$ *such that* $I \sqsubseteq \kappa$*, for all* $\kappa \in \mathcal{S}_0$*.*

Intuitively, objects in $\mathcal{S}$ represent the *kinds* of grades one wants to work with, while the arrows, namely, the order relation, model a refinement between such kinds: $\kappa \sqsubseteq \mu$ means that the kind κ is *more specific* than the kind μ. The operation $\oplus$ combines two kinds to produce the most specific kind generalising both. Finally, the kind I is the most specific one.

It is easy to check that the following properties hold for all $\kappa, \mu, \nu \in \mathcal{S}_0$:

$$(\kappa \oplus \mu) \oplus \nu = \kappa \oplus (\mu \oplus \nu) \qquad \kappa \oplus \kappa = \kappa$$
$$\kappa \oplus \mu = \mu \oplus \kappa \qquad \kappa \oplus I = \kappa$$

namely, $\langle \mathcal{S}_0, \oplus, I \rangle$ is a commutative idempotent monoid.

▶ **Definition 19.** *A* heterogeneous grade algebra *over the grade signature* $\mathcal{S}$ *is just a functor* $H \colon \mathcal{S} \to \mathcal{GrAlg}$*. That is, it consists of a grade algebra* $H(\kappa)$*, written also* H_κ*, for every kind* $\kappa \in \mathcal{S}_0$*, and a grade algebra homomorphism* $H_{\kappa,\mu} \colon H_\kappa \to H_\mu$ *for every arrow* $\kappa \sqsubseteq \mu$*, respecting composition and identities*[8] *, that is,* $\kappa \sqsubseteq \mu \sqsubseteq \nu$ *implies* $H_{\kappa,\nu} = H_{\mu,\nu} \circ H_{\kappa,\mu}$ *and* $H_{\kappa,\kappa} = \mathsf{id}_{H_\kappa}$*.*

Essentially, the homomorphisms $H_{\kappa,\mu}$ realise the refinement $\kappa \sqsubseteq \mu$, transforming grades of kind κ into grades of kind μ, preserving the grade algebra structure.

Observe that the arrows $I \sqsubseteq \kappa$ and $\kappa \sqsubseteq \kappa \oplus \mu$ and $\mu \sqsubseteq \kappa \oplus \mu$ in $\mathcal{S}$ give rise to the following grade algebra homomorphisms:

$$\mathsf{in}^H_\kappa = H_{I,\kappa} \colon H_I \to H_\kappa \quad \mathsf{lh}^H_{\kappa,\mu} = H_{\kappa,\kappa\oplus\mu} \colon H_\kappa \to H_{\kappa\oplus\mu} \quad \mathsf{rh}^H_{\kappa,\mu} = H_{\mu,\kappa\oplus\mu} \colon H_\mu \to H_{\kappa\oplus\mu}$$

[8] The notation $H_{\kappa,\mu}$ makes sense, since between κ and μ there is at most one arrow.

which provide us with a way to map grades of kind I into grades of any other kind, and grades of kind κ and μ into grades of their composition $\kappa \oplus \mu$. By functoriality of H and using the commutative idempotent monoid structure of $\mathcal{S}$, we get the following equalities hold in the category $\mathcal{GrAlg}$, ensuring consistency of such transformations:

$$\mathsf{lh}^H_{\kappa\oplus\mu,\nu} \circ \mathsf{lh}^H_{\kappa,\mu} = \mathsf{lh}^H_{\kappa,\mu\oplus\nu} \tag{1}$$
$$\mathsf{rh}^H_{\kappa\oplus\mu,\nu} \circ \mathsf{lh}^H_{\kappa,\mu} = \mathsf{rh}^H_{\kappa,\mu\oplus\nu} \circ \mathsf{lh}^H_{\mu,\nu} \tag{2}$$
$$\mathsf{lh}^H_{\kappa,\mu} = \mathsf{rh}^H_{\mu,\kappa} \tag{3}$$
$$\mathsf{lh}^H_{\kappa,\kappa} = \mathsf{id}_{H_\kappa} \tag{4}$$
$$\mathsf{lh}^H_{\kappa,I} = \mathsf{id}_{H_\kappa} \tag{5}$$
$$\mathsf{rh}^H_{\kappa,I} = \mathsf{in}^H_\kappa \tag{6}$$

In the following, we will show how to turn a heterogeneous grade algebra into a single grade algebra. The procedure we will describe is based on a general construction due to Grothendieck [17] defined on indexed categories.

Let us assume a grade signature $\mathcal{S}$ and a heterogeneous grade algebra $H \colon \mathcal{S} \to \mathcal{GrAlg}$. We consider the following set:

$$|G(H)| = \{\langle \kappa, r\rangle \mid \kappa \in \mathcal{S}_0,\ r \in |H_\kappa|\}$$

That is, elements of $G(H)$ will be *kinded grades*, namely, pairs of a kind κ and a grade of that kind. Note that this is indeed a set because $\mathcal{S}$ is small, that is, $\mathcal{S}_0$ is a set. Then, we define a binary relation $\preceq_H$ on $|G(H)|$ as follows:

$$\langle \kappa, r\rangle \preceq_H \langle \mu, s\rangle \quad \text{iff} \quad \kappa \sqsubseteq \mu \text{ and } H_{\kappa,\mu}(r) \preceq_\mu s$$

that is, the kind κ must be more specific than the kind μ and, transforming r by $H_{\kappa,\mu}$, we obtain a grade of kind μ which is smaller than s. These data define a partially ordered set as the following proposition shows.

▶ **Proposition 20.** *$\langle |G(H)|, \preceq_H\rangle$ is a partially ordered set.*

The additive structure is given by a binary operation $+_H \colon |G(H)| \times |G(H)| \to |G(H)|$ and an element $\mathbf{0}_H$ in $|G(H)|$ defined as follows:

$$\langle \kappa, r\rangle +_H \langle \mu, s\rangle = \langle \kappa \oplus \mu, \mathsf{lh}^H_{\kappa,\mu}(r) +_{\kappa\oplus\mu} \mathsf{rh}^H_{\kappa,\mu}(s)\rangle \qquad \mathbf{0}_H = \langle I, \mathbf{0}_I\rangle$$

That is, the addition of $\langle \kappa, r\rangle$ and $\langle \mu, s\rangle$ is performed by first mapping r and s in the most specific kind generalising both κ and μ, namely $\kappa \oplus \mu$, and then by summing them in the grade algebra over that kind. The zero element is just the zero of the most specific kind.

▶ **Proposition 21.** *$\langle |G(H)|, \preceq_H, +_H, \mathbf{0}_H\rangle$ is an ordered commutative monoid.*

▶ **Proposition 22.** *$\mathbf{0}_H \preceq_H \langle \kappa, r\rangle$ for every $\langle \kappa, r\rangle \in |G(H)|$.*

Similarly, the multiplicative structure is given by a binary operation $\cdot_H \colon |G(H)| \times |G(H)| \to |G(H)|$ and an element $\mathbf{1}_H$ in $|G(H)|$ defined as follows:

$$\langle \kappa, r\rangle \cdot_H \langle \mu, s\rangle = \begin{cases} \langle \kappa \oplus \mu, \mathsf{lh}^H_{\kappa,\mu}(r) \cdot_{\kappa\oplus\mu} \mathsf{rh}^H_{\kappa,\mu}(s)\rangle & \langle \kappa, r\rangle \neq \mathbf{0}_H \text{ and } \langle \mu, s\rangle \neq \mathbf{0}_H \\ \mathbf{0}_H & \text{otherwise} \end{cases} \qquad \mathbf{1}_H = \langle I, \mathbf{1}_I\rangle$$

Notice that the definitions above follow almost the same pattern as additive operations, but we force that multiplying by $\mathbf{0}_H$ we get again $\mathbf{0}_H$, which is a key property of grade algebras.

▶ **Proposition 23.** *$\langle |G(H)|, \preceq_H, \cdot_H, \mathbf{1}_H \rangle$ is an ordered monoid.*

Altogether, we finally get the following result.

▶ **Theorem 24.** *$G(H) = \langle |G(H)|, \preceq_H, +_H, \cdot_H, \mathbf{0}_H, \mathbf{1}_H \rangle$ is a grade algebra.*

6 Grades as Java expressions

In Section 4 we described how a Java-like language could be equipped with *grades* decorating types, taken in an arbitrary grade algebra. Moreover, in Section 5, we have shown that such grade algebra could have been obtained by composing, in a specific way determined by providing a minimal collection of grade homomorphisms, different grade algebras corresponding to different ways to track usage of resources. In this section, we consider the issue of providing a linguistic support to this aim. This could be done by using an ad-hoc configuration language, however, we believe an interesting solution is that the grade annotations in types could be written themselves in Java.

The key idea is that grade annotations are Java expressions of (classes implementing) a predefined interface `Grade`, analogously to Java exceptions which are expressions of (subclasses of) `Exception`. Moreover, grade homomorphisms are user-defined as well. Namely, a user program can include:

- pairs of *grade classes* and *grade factory classes*, each one modeling a grade algebra desired for the specific application, with the factory class providing its constants
- *grade homomorphism classes*, each one modeling a homomorphism from a grade algebra (class) to another.

When typechecking code with grade annotations, the grades internally used by the type system are those obtained by combining all the declared grade algebras (classes) by means of the declared grade homomorphism classes, with the construction described in Section 5. Operations on grades in the same grade algebra (class) are derived from user-defined methods, as discussed more in detail below, whereas operations on heterogeneous grades are derived as in the construction in Section 5.

Grade and grade factory classes. They are classes implementing the following generic interfaces, respectively:

```
interface Grade<T extends Grade<T>> {
  boolean leq(T x);
  T sum(T x);
  T mult(T x);
}
interface GradeFactory<T extends Grade<T>>{
  T zero();
  T one();
}
```

Grade homomorphism classes. They are classes implementing the following generic interface:

```
interface GradeHom<T extends Grade<T>, R extends Grade<R>> {
  R apply(T x);
}
```

Examples of grade classes and grade homomorphism classes can be found in [4].

Typechecking could then be performed in two steps:

1. Code defining grades, which is assumed to be standard (that is, non-graded) Java code, is typechecked by the standard compiler.
2. Graded code (containing grade annotations written in Java) is typechecked accordingly to the graded type system in Figure 4, where the underlying grade algebra is obtained by composing, by the construction described in Section 5, the user-defined grade algebras through the user-defined grade homomorphisms. Each user-defined algebra has as carrier (set of grades) the Java values which are instances of the corresponding class, and the operations are computed by executing user-defined methods in such class. For instance, to compute the sum $v_1 + v_2$ of two grades which are values of a grade class, we evaluate v_1`.sum(`v_2`)`. Analogously to compute the result of a grade homomorphism.

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

- Grade classes, grade factory classes, and grade homomorphism classes should satisfy the axioms required for the structures they model, e.g., that the sum derived from `sum` methods is commutative and associative. The same happens, for instance, in Haskell, when one defines instances of `Functor` or `Monad`.
- Code defining grades should be *terminating*, since, as described above, the second typechecking step requires to *execute* code typechecked in the first step.
- Finally, the relation among grade classes implicitly defined by declaring grade homomorphism classes should actually be a direct refinement relation, that is, should satisfy the two requirements: (1) there exists at most one path between two grade classes, and (2) each two grade classes with a common ancestor have a *least* common ancestor. These are requirements easy to check, similarly to the check that inheritance is acyclic, or that there are no diamonds in multiple inheritance.

An interesting point is that 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.

7 Related work

The two contributions which have been more inspiring for the work in this paper are the instrumented semantics proposed in [8] and the Granule language [23]. In [8], the authors develop GrAD, a graded dependent type system that includes functions, tensor products, additive sums, and a unit type. Moreover, they define an instrumented operational semantics which tracks usage of resources, and prove that the graded type system is sound with respect to such instrumented semantics. In this paper, we take the same approach to define a resource-aware semantics, parametric on an arbitrary grade algebra. However, differently from [8], where such semantics is defined on typed terms, with the only aim to show the role of the type system, the definition of our semantics is given *independently* from the type system, as is the standard approach in calculi. That is, the aim is also to provide a simple purely semantic model which takes into account usage of resources.

Granule [23] is a functional language equipped with graded modal types, where different kinds of 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 idea of allowing different kinds of grades to coexist, and the overall objective to exploit graded modal types in a programming language. Concerning heterogeneous grades, in this paper we push forward the Granule approach, since we do not want this grade algebra to be fixed, but extendable

by the programmer with user-defined grades. To this aim we define the construction in Section 5. Concerning the design of a graded programming language, here we investigate the object-oriented rather than functional paradigm, taking some solutions which seem more adequate in that context, e.g., to have once-graded types and no boxing/unboxing. The design and implementation of a real Java-like language are not objectives of the current paper; however, we outline in Section 6 a possible interesting solution, where grade annotations are written in the language itself.

Coming more in general to resource-aware type systems, coeffects were first introduced by [24] and further analyzed by [25]. In particular, [25] 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 [7], who developed a framework for structural coeffects for a functional language. This approach is inspired by a generalization of the exponential modality of linear logic, see, e.g., [6]. That is, the distinction between linear and unrestricted variables of linear systems is generalized to have variables decorated by coeffects, or *grades*, that determine how much they can be used. In this setting, many advances have been made to combine coeffects with other programming features, such as computational effects [14, 23, 11], dependent types [2, 8, 22], and polymorphism [1]. Other graded type systems are explored in [2, 15, 1], also combining effects and coeffects [14, 23]. 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).

As already mentioned, [22] and [27] observed that contexts in a structural coeffect system form a module over the semiring of grades, event though they do not use this structure in its full generality, restricting themselves to free modules, that is, to structural coeffect systems. Recently, [5] shows a significant non-structural instance, namely, a coeffect system to track sharing in the imperative paradigm.

8 Conclusion

The contributions of the paper can be summarized as follows:

- Resource-aware extension of FJ reduction, parametric on an arbitrary grade algebra.
- Resource-aware extension of the type system, proved to ensure soundness-may of the resource-aware semantics.
- Formal construction which, given grades of different kinds and grade transformations corresponding to a refinement relation among kinds (formally, a functor over a grade signature), provides a grade algebra of *heterogeneous grades*.
- Notion of *direct refinement* allowing a minimal and easy to check way to specify the above functor.
- Outline of a Java extension where grades are user-defined, and grade annotations are written in the language itself.

As already noted, the key novel ideas in the contributions above are mostly independent from the language. So, a first natural direction for future work is to explore their incarnation in another paradigm, e.g., the functional one. That would include the definition of a parametric

resource-aware reduction independent from types, the design of a type system with once-graded types, and possibly the design of user-defined grades in a functional language, e.g., in Haskell by relying on the typeclass feature. Though the overall approach should still apply, we expect the investigation to be significant due to the specific features of the paradigm.

The resource-aware operational semantics defined in this paper requires *annotations* in subterms, with the only aim to fix their reduction grade in the reduction of the enclosing term. As mentioned in Section 3, adopting a big-step style would clearly remove the need of such technical artifice; only annotations in constructor subterms should be kept, since they express a true constraint on the semantics. Thus, a very interesting alternative to be studied is a big-step version of resource-aware semantics, allowing a more abstract and clean presentation. With this choice, we should employ, to prove soundness-may, the techniques recently introduced in [10, 9].

Coming back to Java-like languages, the FJ language considered in the paper does not include imperative features. Adding mutable memory leads to many significant research directions. First, besides the model presented in this paper, and in general in literature, where "using a resource" means "replacing a variable with its value", another view is possible where the resource is the memory and "using" means "interacting with the memory". Moreover, we would like to investigate more in detail how to express by grade algebras forms of usages which are typical of the imperative paradigm, such as the readonly modifier, and, more in general, *capabilities* [18, 16].

References

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

2 Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 56–65. ACM Press, 2018. doi:10.1145/3209108.3209189.

3 Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. A Java-like calculus with user-defined coeffects. In Ugo Dal Lago and Daniele Gorla, editors, *ICTCS'22 – Italian Conference on Theoretical Computer Science*, volume 3284 of *CEUR Workshop Proceedings*, pages 66–78. CEUR-WS.org, 2022.

4 Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. Multi-graded Featherweight Java. *CoRR*, 2023. URL: http://arxiv.org/abs/2302.07782.

5 Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, and Marco Servetto. Coeffects for sharing and mutation. *Proceedings of ACM on Programming Languages*, 6(OOPSLA):870–898, 2022. doi:10.1145/3563319.

6 Flavien Breuvart and Michele Pagani. Modelling coeffects in the relational semantics of linear logic. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic, CSL 2015*, volume 41 of *LIPIcs*, pages 567–581. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.CSL.2015.567.

7 Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In Zhong Shao, editor, *European Symposium on Programming, ESOP 2013*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer, 2014. doi:10.1007/978-3-642-54833-8_19.

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

9 Francesco Dagnino. A meta-theory for big-step semantics. *ACM Transactions on Computational Logic*, 23(3):20:1–20:50, 2022. doi:10.1145/3522729.

10 Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness conditions for big-step semantics. In Peter Müller, editor, *European Symposium on Programming, ESOP 2020*, volume 12075 of *Lecture Notes in Computer Science*, pages 169–196. Springer, 2020. doi:10.1007/978-3-030-44914-8_7.

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

12 Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1):83–133, 1984. doi:10.1016/0304-3975(84)90113-0.

13 Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In Erik Ernst, editor, *European Conference on Object-Oriented Programming, ECOOP 2007*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer, 2007.

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

15 Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In Zhong Shao, editor, *European Symposium on Programming, ESOP 2013*, volume 8410 of *Lecture Notes in Computer Science*, pages 331–350. Springer, 2014. doi:10.1007/978-3-642-54833-8_18.

16 Colin S. Gordon. Designing with static capabilities and effects: Use, mention, and invariants (pearl). In Robert Hirschfeld and Tobias Pape, editors, *European Conference on Object-Oriented Programming, ECOOP 2020*, volume 166 of *LIPIcs*, pages 10:1–10:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ECOOP.2020.10.

17 Alexander Grothendieck. Catégories fibrées et descente. In *Revêtements étales et groupe fondamental*, pages 145–194. Springer, 1971.

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

19 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146. ACM Press, 1999. doi:10.1145/320384.320395.

20 Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

21 Daniel Marshall, Michael Vollmer, and Dominic Orchard. Linearity and uniqueness: An entente cordiale. In Ilya Sergey, editor, *European Symposium on Programming, ESOP 2022*, volume 13240 of *Lecture Notes in Computer Science*, pages 346–375. Springer, 2022. doi:10.1007/978-3-030-99336-8_13.

22 Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World – Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.

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

24 Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: Unified static analysis of context-dependence. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages and Programming, ICALP 2013*, volume 7966 of *Lecture Notes in Computer Science*, pages 385–397. Springer, 2013. doi:10.1007/978-3-642-39212-2_35.

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

26 Emily Riehl. *Category theory in context*. Courier Dover Publications, 2017.

27 James Wood and Robert Atkey. A framework for substructural type systems. In Ilya Sergey, editor, *European Symposium on Programming, ESOP 2022*, volume 13240 of *Lecture Notes in Computer Science*, pages 376–402. Springer, 2022. doi:10.1007/978-3-030-99336-8_14.