Resource-Aware Soundness for Big-Step Semantics

RICCARDO BIANCHINI, University of Genoa, Italy
FRANCESCO DAGNINO, University of Genoa, Italy
PAOLA GIANNINI, University of Eastern Piedmont, Italy
ELENA ZUCCA, University of Genoa, Italy

We extend the semantics and type system of a lambda calculus equipped with common constructs to be *resource-aware*. That is, reduction is instrumented to keep track of the usage of resources, and the type system guarantees, besides standard soundness, that for well-typed programs there is a computation where no needed resource gets exhausted. The resource-aware extension is parametric on an arbitrary *grade algebra*, and does not require ad-hoc changes to the underlying language. To this end, the semantics needs to be formalized in big-step style; as a consequence, expressing and proving (resource-aware) soundness is challenging, and is achieved by applying recent techniques based on coinductive reasoning.

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

Additional Key Words and Phrases: Graded modal types, generalized inference systems

ACM Reference Format:
Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2023. Resource-Aware Soundness for Big-Step Semantics. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 267 (October 2023), 29 pages. https://doi.org/10.1145/3622843

1 INTRODUCTION

An increasing interest has been devoted in recent research to *resource-awareness*, that is, to formal techniques for reasoning about the usage of resources in computations, where different kinds of usage are modeled as *grades*. For instance, taking grades of shape $r ::= 0 \mid 1 \mid \omega$, meaning either not used, or used once[1], or used in an unrestricted way, we can distinguish the functions $\lambda x.5$, $\lambda x.x$, and $\lambda x.x + x$, by assigning the grade 0, 1, and ω, respectively, to their parameter. A similar example is counting; that is, grades are natural numbers with either bounded or exact usage; however, grades can equally well model non-quantitative information, e.g., the fact that the parameter of a function is used with a given privacy level. In the different proposals in the 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*.

Whereas most literature has been devoted to *resource-aware type systems*, where grades are used as annotations of types[2] [Atkey 2018; Brunel et al. 2014; Dal Lago and Gavazzo 2022; Gaboardi et al. 2016; Ghica and Smith 2014; Orchard et al. 2019; Petricek et al. 2014], a few works have considered *resource-aware semantics* as well [Bianchini et al. 2023b; Choudhury et al. 2021], where evaluation

[1]In the flavour either "exactly" or "at most" once.
[2]Such annotated types are also called *graded modal types*.

Authors' addresses: Riccardo Bianchini, riccardo.bianchini@edu.unige.it, University of Genoa, , Italy; Francesco Dagnino, francesco.dagnino@dibris.unige.it, University of Genoa, , Italy; Paola Giannini, paola.giannini@uniupo.it, University of Eastern Piedmont, , Italy; Elena Zucca, elena.zucca@unige.it, University of Genoa, , Italy.

2475-1421/2023/10-ART267
https://doi.org/10.1145/3622843

takes place in an *enviroment* of available resources, each one with an associated grade modeling its current amount. Since both the semantics and the type system are graded, we can formally state *resource-aware soundness*, meaning that well-typed programs can neither get stuck due to standard typing errors, nor due to resource exhaustion.

The aim of this paper is to design resource-aware semantics and a type system, and formulate and prove resource-aware soundness, in as much as possible a light, abstract and general way, without requiring ad-hoc changes to the underlying language. We detail below how this is achieved, pointing out novelties with respect to previous work.

Keeping original syntax. In the literature on graded type systems, the production of types has shape $T ::= \ldots \mid T^r$, 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 by r, and a corresponding unboxing mechanism. Here, we take a much lighter approach, where the syntax of terms is not affected, in adherence to the "no ad-hoc changes" principle mentioned above. The production for types is $T ::= \tau^r$, with τ non-graded type, i.e., all types are (once) graded. Since there is no boxing/unboxing, there is no explicit promotion rule, but different grades can be assigned to an expression, assuming different contexts. To our knowledge, such an approach with no boxing/unboxing has only been previously adopted in [Bianchini et al. 2023b], where, however, a Java-like language was considered, hence with a much simpler type system, nominal and with no functional types.

Paradigmatic underlying language. We add resource-awareness on top of an extended lambda calculus, intended to be representative of typical language features. Most literature on graded type systems considers similar lambda calculi; however, besides the fact that we do not add ad-hoc syntax, as discussed above, another distinguishing novelty of our work is that we include in the calculus two important constructs which are only marginally considered in the literature. First, we provide an in-depth investigation of resource consumption in recursive functions. Roughly, the declaration of a function adds to the environment a resource which needs to be graded so as to cover the possibile recursive calls; correspondingly, a function which is recursive needs to be typed with an "infinite" grade. Secondly, our graded type system smoothly includes *equi-recursive* types, a feature never handled, to our knowledge, in previous work on grades, and which, differently from iso-recursive presentations, once again permits no syntax overhead.

Natural extension of the standard semantics. Since the reduction relation itself is graded[3], in small-step semantics subterms need to be annotated [Bianchini et al. 2023b; Choudhury et al. 2021] to ensure that their reduction happens at each step with the same grade. The novel idea in this paper is to define the resource-aware semantics, instead, in *big-step* style, so that no annotations are needed, again keeping the underlying language unaffected. A consequence of this choice is that proving and even expressing (resource-aware) soundness of the type system becomes challenging, since in big-step semantics non-terminating and stuck computations are indistinguishable [Cousot and Cousot 1992; Leroy and Grall 2009]. To solve this problem, the big-step judgment is extended to model divergence explicitly, and is defined by a *generalized inference system* [Ancona et al. 2017a; Dagnino 2019], where rules are interpreted in an *essentially coinductive*, rather than inductive, way, in the sense that infinite proof trees are allowed, in a controlled way. We note that our proof of resource-aware soundness is a significant application of these innovative techniques.

Parametricity. The resource-aware semantics and type system are *parametric* on an arbitrary grade algebra. Such independence from the specific nature of grades is common to most, but not

[3]To express that its final result will be a value to be used with a given grade.

all, approaches. In particular, compared to [Bianchini et al. 2023b] whose type system, like ours, supports no box/unbox constructs, an important novelty is that here we extend this approach to a more general definition of grade algebra, including *non-affine* instances as well, for instance, natural numbers with exact counting. It is worthwhile to note that, when the underlying algebra is non-affine, some language constructs may turn out to be ill-typed, since resources are not consumed in an exact way, as we will discuss in Section 6.

The "no ad-hoc changes" principle is achieved in two respects. First, the syntax is unaffected. This is very important from a language design point of view, meaning that it would be possible in principle to add resource-awareness to an arbitrary language, along the lines shown here:

- without requiring the programmer to learn new non-trivial constructs (e.g., box/unbox could be hard for a Java programmer)
- ensuring backward compatibility, since old code will still work, by only embedding plain types into graded types. This is always possibile; for instance, with grades $r ::= 0 \mid 1 \mid \omega$, non-graded code can simply be seen as ω-graded.

Moreover, the semantics does not need annotations of subterms. This is a more technical feature, unimportant for the standard programmer, but allowing a cleaner and simpler way to analyse the behaviour of programs, notably reasoning directly on source code.

In a nutshell, we illustrate how to equip an arbitrary language with resource-awareness, keeping syntax unaffected, and proving resource-aware soundness. Most previous works on graded type systems do not consider resource-aware soundness, and require constructs for explicit promotion (box/unbox); [Choudhury et al. 2021] introduces resource-aware semantics and soundness, but still has box/unbox, and does not include recursion/non-termination and recursive types; [Bianchini et al. 2023b] has resource-aware semantics and soundness and keeps syntax unaffected, but, taking a Java-like language, does not deal with higher-order functions and only has nominal types. The calculus in our paper includes recursion, higher-order functions, and recursive types, hence can be taken as a powerful enough example to illustrate how to add resource-awareness.

After formally defining grade algebras in Section 2, in Section 3 and Section 4 we present resource-aware reduction, and type system, respectively. In Section 5 we prove resource-aware soundness. We provide examples and discussions in Section 6. Section 7 surveys related work, summarizes the contributions, and outlines future work. Proofs of lemmas can be found in Appendix A.

2 ALGEBRAIC PRELIMINARIES: A TAXONOMY OF GRADE ALGEBRAS

In this section we introduce the algebraic structures we will use throughout the paper. At the core of our work there are *grades*, namely, annotations in terms and types expressing how or how much resources can be used during the computation. As we will see, we need some operations and relations to properly combine and compare grades in the resource-aware semantics and type system, hence we assume grades to form an algebraic structure called *grade algebra* defined below. Such a structure is a slight variant of others in literature [Abel and Bernardy 2020; Atkey 2018; Brunel et al. 2014; Choudhury et al. 2021; Gaboardi et al. 2016; Ghica and Smith 2014; McBride 2016; Orchard et al. 2019; Wood and Atkey 2022], which are all instances of ordered semirings.

Definition 2.1 (Ordered Semiring). An *ordered semiring* 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 $r, r's, s' \in |R|$;

Essentially, an ordered semiring is a semiring with a partial order on its carrier which makes addition and multiplication monotonic with respect to it. Roughly, addition and multiplication (which is not necessarily commutative) provide parallel and sequential composition of usages, $\mathbf{1}$ models the unitary or default usage and $\mathbf{0}$ models no use. Finally, the partial order models overapproximation in the resource usage, which allows for flexibility, for instance we can have different usage in the branches of an if-then-else construct.

In an ordered semiring there can be elements $r \preceq \mathbf{0}$, which, however, make no sense in our context, since $\mathbf{0}$ models no use. Hence, in a grade algebra we forbid such grades.

Definition 2.2 (Grade Algebra). An ordered semiring $R = \langle |R|, \preceq, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$ is a *grade algebra* if $r \preceq \mathbf{0}$ implies $r = \mathbf{0}$, for all $r \in |R|$.

This property is not technically needed, but motivated by modelling reasons and to simplify some definitions. Moreover, it can be forced in any ordered semiring, just noting that the set $I_{\preceq \mathbf{0}} = \{r \in |R| \mid r \preceq \mathbf{0}\}$ is a two-sided ideal and so the quotient semiring $R/I_{\preceq \mathbf{0}}$ is well-defined and is a grade algebra. We now give some examples of grade algebras adapted from the literature.

Example 2.3.

(1) The simplest way of measuring resource usage is by counting, as can be done using natural numbers with their usual operations. We consider two grade algebras over natural numbers: one for *bounded usage* $\mathsf{Nat}^{\leq} = \langle \mathbb{N}, \leq, +, \cdot, 0, 1 \rangle$ taking the natural ordering and another for *exact usage* $\mathsf{Nat}^{=} = \langle \mathbb{N}, =, +, \cdot, 0, 1 \rangle$ taking equality as order, thus basically forbidding approximations of resource usage.

(2) The *linearity* grade algebra $\langle \{0, 1, \infty\}, \leq, +, \cdot, 0, 1 \rangle\}$ is obtained from $\mathsf{Nat}^{=}$ above by identifying all natural numbers strictly greater than 1 and taking as order $0 \leq \infty$ and $1 \leq \infty$; the *affinity* grade algebra only differs for the order, which is $0 \leq 1 \leq \infty$.

(3) In the trivial grade algebra Triv the carrier is a singleton set $|\mathsf{Triv}| = \{\infty\}$, the partial order is the equality, addition and multiplication are defined in the trivial way and $\mathbf{0}_{\mathsf{Triv}} = \mathbf{1}_{\mathsf{Triv}} = \infty$.

(4) The grade algebra of extended non-negative real numbers is the tuple $\mathsf{R}^{\infty}_{\geq 0} = \langle [0, \infty], \leq, +, \cdot, 0, 1 \rangle$, where usual order and operations are extended to ∞ in the expected way.

(5) A distributive lattice $\mathsf{L} = \langle |\mathsf{L}|, \leq, \vee, \wedge, \bot, \top \rangle$, where $\leq$ is the order, $\vee$ and $\wedge$ the join and meet, and $\bot$ and $\top$ the bottom and top element, respectively, is a grade algebra. Such grade algebras do not carry a quantitative information, as the addition is idempotent, but rather express how/in which mode a resource can be used. They are called *informational* by [Abel and Bernardy 2020].

(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* $R \times S = \langle \{(r, s) \mid r \in |R| \wedge s \in |S|\}, \preceq, +, \cdot, (\mathbf{0}_R, \mathbf{0}_S), (\mathbf{1}_R, \mathbf{1}_S) \rangle$, where operations are the pairwise application of the operations for R and S, is a grade algebra.

(7) Given a grade algebra $R = \langle |R|, \preceq_R, +_R, \cdot_R, \mathbf{0}_R, \mathbf{1}_R \rangle$, set $R^{\infty} = \langle |R| + \{\infty\}, \preceq, +, \cdot, \mathbf{0}_R, \mathbf{1}_R \rangle$ where $\preceq$ extends $\preceq_R$ by adding $r \preceq \infty$ for all $r \in |R^{\infty}|$ and $+$ and $\cdot$ extend $+_R$ and $\cdot_R$ by $r + \infty = \infty + r = \infty$, for all $r \in |R^{\infty}|$, and $r \cdot \infty = \infty \cdot r = \infty$, for all $r \in |R^{\infty}|$ with $r \neq \mathbf{0}_R$, and $\mathbf{0}_R \cdot \infty = \infty \cdot \mathbf{0}_R = \mathbf{0}_R$. Then, R^{∞} is a grade algebra, where ∞ models *unrestricted usage*.

(8) Given R as above, set $|\mathsf{Int}(R)| = \{\langle r, s \rangle \in |R| \times |R| \mid r \preceq_R s\}$, the set of *intervals* between two points in $|R|$, with $\langle r, s \rangle \preceq \langle r', s' \rangle$ iff $r' \preceq_R r$ and $s \preceq_R s'$. Then, $\mathsf{Int}(R) = \langle |\mathsf{Int}(R)|, \preceq, +, \cdot, \langle \mathbf{0}_R, \mathbf{0}_R \rangle, \langle \mathbf{1}_R, \mathbf{1}_R \rangle \rangle$ is a grade algebra, with $+$ and $\cdot$ defined pointwise.

We say that a grade algebra is *trivial* if it is isomorphic to Triv, that is, it contains a single point. It is easy to see that a grade algebra is trivial iff $\mathbf{1} \preceq \mathbf{0}$, as this implies $r \preceq \mathbf{0}$, hence $r = \mathbf{0}$, for all $r \in |R|$, by the axioms of ordered semiring and Definition 2.2.

Remark 2.4. In a grade algebra R, grades $r \in |R|$ satisfying $\mathbf{0} \preceq r$ play a special role: they represent usages that can be *discarded*, since they can be reduced to $\mathbf{0}$ through the approximation order. For instance, in the linearity grade algebra of Example 2.3(2), the element ∞ can be discarded, while the element 1 cannot. A grade algebra is said *affine* if $\mathbf{0} \preceq r$ holds for all $r \in |R|$. Note that this condition is equivalent to requiring just $\mathbf{0} \preceq \mathbf{1}$ again thanks to the axioms of ordered semiring. Instances of affine grade algebras from Example 2.3 are bounded usage (Item 1) and distributive lattices (Item 5), while exact usage (Item 1) and linearity (Item 2) are not affine.

A grade algebra is still a quite wild structure, notably there are weird phenomena due to the interaction of addition and multiplication with zero. In particular, we can get $\mathbf{0}$ by summing or multiplying non-zero grades, that is, there are relevant usages that when combined elide each other.

Definition 2.5. Let $R = \langle |R|, \preceq, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$ be a grade algebra. We say that

- R is *integral* if $r \cdot s = \mathbf{0}$ implies $r = \mathbf{0}$ or $s = \mathbf{0}$, for all $r, s \in |R|$;
- R is *reduced* if $r + s = \mathbf{0}$ implies $r = s = \mathbf{0}$, for all $r, s \in |R|$.

All grade algebras in Example 2.3 are reduced, provided that the parameters are reduced as well. Similarly, they are all integral except Items 5 and 6. Indeed, in the former there can be elements different from $\bot$ whose meet is $\bot$ (e.g., disjoint subsets in the powerset lattice), while in the latter there are "spurious" pairs $\langle r, \mathbf{0} \rangle$ and $\langle \mathbf{0}, s \rangle$ whose product is $\langle \mathbf{0}, \mathbf{0} \rangle$ even if both $r \neq \mathbf{0}$ and $s \neq \mathbf{0}$. Fortunately, there is an easy construction making a grade algebra both reduced and integral.

Definition 2.6. Set $R = \langle |R|, \preceq, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$ an ordered semiring. We denote by $R_{\mathbf{0}}$ the ordered semiring $\langle |R| + \{\hat{\mathbf{0}}\}, \preceq, +, \cdot, \hat{\mathbf{0}}, \mathbf{1} \rangle$ where we add a new point $\hat{\mathbf{0}}$ and extend order and operations as follows:

$$\hat{\mathbf{0}} \preceq r \text{ iff } \mathbf{0} \preceq r,$$
$$r + \hat{\mathbf{0}} = \hat{\mathbf{0}} + r = r,$$
$$r \cdot \hat{\mathbf{0}} = \hat{\mathbf{0}} \cdot r = \hat{\mathbf{0}}, \text{ for all } r \in |R| + \{\hat{\mathbf{0}}\}.$$

It is easy to check that the following proposition holds.

Proposition 2.7. *If $R = \langle |R|, \preceq, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$ is a grade algebra, then $R_{\mathbf{0}}$ is a reduced and integral grade algebra.*

Applying this construction to Items 5 and 6 we get reduced and integral grade algebras. However, for the latter the result is not yet satisfactory. Indeed, the resulting grade algebra still has spurious elements which are difficult to interpret. Thus we consider the following refined construction.

Example 2.8. Let R and S be non-trivial, reduced and integral grade algebras. The *smash product* of R and S is $R \owedge S = \langle |R \owedge S|, \preceq, +, \cdot, \hat{\mathbf{0}}, \langle \mathbf{1}_R, \mathbf{1}_S \rangle \rangle$, where $|R \owedge S| = |(R \times S)_{\mathbf{0}}| \setminus (|R| \times \{\mathbf{0}_S\} \cup \{\mathbf{0}_R\} \times |S|)$, $\preceq$, $+$ and $\cdot$ are the restrictions of the order and operations of $(R \times S)_{\mathbf{0}}$, as in Definition 2.6, to the subset $|R \owedge S|$, and $\hat{\mathbf{0}}$ is the zero of $(R \times S)_{\mathbf{0}}$. It is easy to see $R \owedge S$ is a non-trivial, reduced and integral grade algebra.

In the rest of the paper we will assume an integral grade algebra $R = \langle |R|, \preceq, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$. Requiring R to be integral allows some simplifications, in particular, this ensures that multiplying non-zero grades we cannot get $\mathbf{0}$, as useful, e.g., in Lemma 4.2(2).

3 RESOURCE-AWARE SEMANTICS

We define, for a standard functional calculus, an instrumented semantics which keeps track of resource usage, hence, in particular, it gets stuck if some needed resource is insufficient.

$$
\begin{array}{rcll}
e & ::= & x \mid \mathtt{rec}\, f.\lambda x.e \mid e_1 e_2 \mid & \text{expression} \\
& \mid & \mathtt{unit} \mid \mathtt{match}\ e_1\ \mathtt{with}\ \mathtt{unit}\ \longrightarrow e_2 & \\
& \mid & \langle {}^r e_1, e_2{}^s \rangle \mid \mathtt{match}\ e_1\ \mathtt{with}\ \langle x, y \rangle \longrightarrow e_2 \mid & \\
& \mid & \mathtt{inl}^r e \mid \mathtt{inr}^r e \mid \mathtt{match}\ e\ \mathtt{with}\ \mathtt{inl}\ x_1 \longrightarrow e_1\ \mathtt{or}\ \mathtt{inr}\ x_2 \longrightarrow e_2 \mid & \\
\mathbf{v} & ::= & \mathtt{rec}\, f.\lambda x.e \mid \mathtt{unit} \mid \langle {}^r \mathbf{v}_1, \mathbf{v}_2{}^s \rangle \mid \mathtt{inl}^r \mathbf{v} \mid \mathtt{inr}^r \mathbf{v} & \text{value} \\
\rho & ::= & x_1 : (r_1, \mathbf{v}_1), \ldots, x_n : (r_n, \mathbf{v}_n) & \text{environment}
\end{array}
$$

Fig. 1. Syntax

Surface syntax. The (surface) syntax is given in Fig. 1. We assume variables x, y, f, ..., where the last will be used for variables denoting functions. The constructs are pretty standard: the `unit` constant, pairs, left and right injections, and three variants of `match` construct playing as destructors of units, pairs, and injections, respectively. Instead of standard lambda expressions and a `fix` operator for recursion, we have a unique construct $\mathtt{rec}\, f.\lambda x.e$, meaning a function with parameter x and body e which can recursively call itself through the variable f. Standard lambda expressions can be recovered as those where f does not occur free in e, that is, when the function is non-recursive, and we will use the abbreviation $\lambda x.e$ for such expressions. The motivation for such unique construct is that in the resource-aware semantics there is no immediate parallel substitution as in standard rules for application and `fix`, but occurrences of free variables are replaced one at a time, when needed, by their value stored in an environment. Thus, application of a (possibly recursive) function can be nicely modeled by generalizing what expected for a non-recursive one, that is, it leads to the evaluation of the body in an environment where both f and x are added as resources, as formalized in rule (APP) in Fig. 5.

The pair and injection constructors are decorated with a grade for each subterm, intuitively meaning "how many copies" are contained in the compound term. For instance, taking as grades the natural numbers as in Example 2.3(1), a pair of shape $\langle {}^2 e_1, e_2{}^2 \rangle$ contains "two copies" of each component. In the resource-aware semantics, this is reflected by the fact that, to evaluate (one copy of) such pair, we need to obtain 2 copies of the results of e_1 and e_2; correspondingly, when matching such result with a pair of variables, both are made available in the environment with grade 2.

We will sometimes use, rather than $\mathtt{match}\ e_1\ \mathtt{with}\ \mathtt{unit}\ \longrightarrow e_2$, the alternative syntax $e_1 ; e_2$, emphasising that there is a sequential evaluation of the two subterms.

Resource-aware semantics by examples. The resource-aware semantics is defined on *configurations*, that is, pairs $e|\rho$ where ρ is an *environment* keeping track of the existing resources, parametrically on a given grade algebra. More precisely, as shown in Fig. 1, the environment is a finite map associating to each resource (variable), besides its value, a grade modeling its *allowed usage*.

The judgment has shape $e|\rho \Rightarrow_r \mathbf{v}|\rho'$, meaning that the configuration $e|\rho$ produces a value $\mathbf{v}$ and a final environment ρ'. The reduction relation is *graded*, that is, indexed by a grade r, meaning that the resulting value can be used (at most) r times, or, in more general (non-quantitative) terms, (at most) in r mode. The grade of a variable in the environment decreases, each time the variable is used, of the amount specified in the reduction grade[4]. Of course, this can only happen if the current grade of the variable *can* be reduced of such an amount. Otherwise, evaluation is stuck; formally, since the reduction relation is big-step, no judgment can be derived.

The choice of big-step style is motivated since small-step style, as in [Bianchini et al. 2023b], needs a syntax where *all* operators have a grade annotation for each subterm, to ensure that all

[4]More precisely, the reduction grade acts as a lower bound for this amount, see comment to rule (VAR).

$$\dfrac{\dfrac{}{\mathsf{p}|\rho \Rightarrow \mathbf{v}|\mathsf{p} : (0, \mathbf{v})}\,(\textsc{var}) \qquad \dfrac{\dots}{\langle \mathsf{u}, \mathsf{u}\rangle|\rho' \Rightarrow \langle \mathsf{u}, \mathsf{u}\rangle|\rho'}\,(\textsc{pair})}{\texttt{match}\ \mathsf{p}\ \texttt{with}\ \langle \mathsf{x}, \mathsf{y}\rangle \to \langle \mathsf{u}, \mathsf{u}\rangle|\rho \Rightarrow \langle \mathsf{u}, \mathsf{u}\rangle|\rho'}\,(\textsc{match-p}) \qquad \begin{array}{l}\rho = \mathsf{p} : (1, \mathbf{v}) \text{ with } \mathbf{v} = \langle \mathbf{v}_1, \mathbf{v}_2\rangle \\ \rho' = \mathsf{p} : (0, \mathbf{v}), \mathsf{x} : (1, \mathbf{v}_1), \mathsf{y} : (1, \mathbf{v}_2) \\ \rho'' = \mathsf{p} : (0, \mathbf{v}), \mathsf{x} : (0, \mathbf{v}_1), \mathsf{y} : (1, \mathbf{v}_2)\end{array}$$

$$\dfrac{\dfrac{}{\mathsf{p}|\rho \Rightarrow \mathbf{v}|\mathsf{p} : (0, \mathbf{v})}\,(\textsc{var}) \qquad \dfrac{\dfrac{}{\mathsf{x}|\rho' \Rightarrow \mathbf{v}_1|\rho''}\,(\textsc{var}) \qquad \dfrac{}{\mathsf{x}|\rho'' \Rightarrow ???}\,(\textsc{var})}{\langle \mathsf{x}, \mathsf{x}\rangle|\rho' \Rightarrow ???}\,(\textsc{pair})}{\texttt{match}\ \mathsf{p}\ \texttt{with}\ \langle \mathsf{x}, \mathsf{y}\rangle \to \langle \mathsf{x}, \mathsf{x}\rangle|\rho \Rightarrow ???}\,(\textsc{match-p})$$

Fig. 2. Examples of resource-aware evaluation (counting usages)

reduction steps have the same grade. Here, instead, as shown above, only operators which model "data containers" (pair and injection constructors) are decorated with grades for their components.

The instrumented semantics will be formally defined on a *fine-grained* version of expressions. However, in order to focus on the key ideas of resource-aware evaluation, first we illustrate its expected behaviour on some simple examples in the surface syntax.

Example 3.1. Let us consider the following expressions:

- $e_1 = \texttt{match}\ \mathsf{p}\ \texttt{with}\ \langle \mathsf{x}, \mathsf{y}\rangle \to \langle \texttt{unit}, \texttt{unit}\rangle$
- $e_2 = \texttt{match}\ \mathsf{p}\ \texttt{with}\ \langle \mathsf{x}, \mathsf{y}\rangle \to \langle \mathsf{x}, \texttt{unit}\rangle$
- $e_3 = \texttt{match}\ \mathsf{p}\ \texttt{with}\ \langle \mathsf{x}, \mathsf{y}\rangle \to \langle \mathsf{x}, \mathsf{y}\rangle$
- $e_4 = \texttt{match}\ \mathsf{p}\ \texttt{with}\ \langle \mathsf{x}, \mathsf{y}\rangle \to \langle \mathsf{x}, \mathsf{x}\rangle$

to be evaluated in the environment $\rho = \mathsf{p} : (1, \mathbf{v})$ with $\mathbf{v} = \langle \mathbf{v}_1, \mathbf{v}_2\rangle$. Assume, first, that grades are natural numbers, see Example 2.3(1). In order to lighten the notation, 1 annotations are considered default, hence omitted. Moreover, in figures we abbreviate `unit` by u. In the first proof tree in Fig. 2 we show the evaluation of e_1. The resource p is consumed, and its available amount (1) is "transferred" to both the resources x and y, which are added in the environment[5], and not consumed.

The evaluation of e_2 is similar, apart that the resource x is consumed as well, and the evaluation of e_3 consumes all resources. Finally, the evaluation of e_4 is stuck, that is, no proof tree can be constructed: indeed, when the second occurrence of x is found, the resource is exhausted, as shown in the second (incomplete) proof tree in Fig. 2. A result could be obtained, instead, if the original grade of p was greater than 1 (e.g., 2), since in this case x (and y) would be added with grade 2, or, alternatively, if the value associated to p in the environment was, e.g., $\mathbf{v} = \langle {}^2\mathbf{v}_1, \mathbf{v}_2\rangle$.

In the example above, grades model *how many times* resources are used. Assume now a grade algebra where grades model a *non-quantitative* knowledge, that is, track possible *modes* in which a resource can be used. A very simple example are privacy levels $0 \preceq \text{private} \preceq \text{public}$. 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. Note that exactly the same structure could be used to model, e.g., modifiers readonly and mutable in an imperative setting, rather than privacy levels. Moreover, the structure can be generalized by adding a 0 element to any distributive lattice as in Example 2.3(5).

Example 3.2. In Example 3.1, writing priv and pub (default, omitted in the annotations) for short, we have, e.g., for $\rho = \mathsf{p} : (\text{pub}, \mathbf{v})$ with $\mathbf{v} = \langle {}^{\text{priv}}\mathbf{v}_1, \mathbf{v}_2\rangle$, that the evaluation in mode pub of e_1 is analogous to that in Fig. 2; however, the evaluation in mode pub of e_2, e_3, and e_4 is stuck, since it

[5]Modulo renaming, omitted here for simplicity.

$$\dfrac{\dfrac{}{\mathsf{p}|\rho \Rightarrow \mathbf{v}|\mathsf{p}:(\mathsf{pub},\mathbf{v})} \qquad \dfrac{\dfrac{}{\mathsf{x}|\rho' \Rightarrow ???}}{\langle \mathsf{x},\mathsf{u}\rangle|\rho' \Rightarrow ???}}{\texttt{match p with } \langle \mathsf{x},\mathsf{y}\rangle \rightarrow \langle \mathsf{x},\mathsf{u}\rangle|\rho \Rightarrow ???} \qquad \begin{array}{l}\rho = \mathsf{p} : (\mathsf{pub},\mathbf{v}) \text{ with } \mathbf{v} = \langle^{\mathsf{priv}}\mathbf{v}_1, \mathbf{v}_2\rangle \\ \rho' = \mathsf{p} : (\mathsf{pub},\mathbf{v}), \mathsf{x} : (\mathsf{priv},\mathbf{v}_1), \mathsf{y} : (\mathsf{pub},\mathbf{v}_2)\end{array}$$

$$\dfrac{\dfrac{}{\mathsf{p}|\rho \Rightarrow \mathbf{v}|\mathsf{p}:(\mathsf{pub},\mathbf{v})} \qquad \dfrac{\dfrac{}{\mathsf{x}|\rho' \Rightarrow_{\mathsf{priv}} \mathbf{v}_1|\rho'} \qquad \dfrac{}{\mathsf{y}|\rho' \Rightarrow_{\mathsf{priv}} \mathbf{v}_2|\rho'}}{\langle \mathsf{x},\mathsf{y}\rangle|\rho' \Rightarrow_{\mathsf{priv}} \langle \mathbf{v}_1,\mathbf{v}_2\rangle|\rho'}}{\texttt{match p with } \langle \mathsf{x},\mathsf{y}\rangle \rightarrow \langle \mathsf{x},\mathsf{y}\rangle|\rho \Rightarrow_{\mathsf{priv}} \langle \mathbf{v}_1,\mathbf{v}_2\rangle|\rho'}$$

Fig. 3. Examples of resource-aware evaluation (privacy levels)

$$\begin{array}{lcll} v & ::= & x \mid \texttt{rec}\, f.\lambda x.e \mid \texttt{unit} \mid \langle {}^r v_1, v_2{}^s\rangle \mid \texttt{inl}^r\, v \mid \texttt{inr}^r\, v & \text{value expression} \\ e & ::= & \texttt{return}\ v \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid v_1\, v_2 & \text{(possibly diverging) expression} \\ & \mid & \texttt{match}\ v\ \texttt{with unit} \rightarrow e & \\ & \mid & \texttt{match}\ v\ \texttt{with}\ \langle x, y\rangle \rightarrow e & \\ & \mid & \texttt{match}\ v\ \texttt{with inl}\ x_1 \rightarrow e_1\ \texttt{or inr}\ x_2 \rightarrow e_2 & \\ \\ c & ::= & e|\rho & \text{configuration} \end{array}$$

Fig. 4. Fine-grained syntax

needs to use the resource x, which gets a grade priv = priv·pub, hence cannot be used in mode pub since pub $\not\preceq$ priv, as we show in the first (incomplete) proof tree for e_2 in Fig. 3. On the other hand, evaluation in mode priv can be safely performed; indeed, resource p can be used in mode priv since priv $\preceq$ pub, as shown in the second proof tree in Fig. 3.

Formal definition of resource-aware semantics. As anticipated, rules defining the instrumented semantics are given on a fine-grained version of the language. This long-standing approach [Levy et al. 2003] is used to clearly separate effect-free from effectful expressions (*computations*), and to make the evaluation strategy, relevant for the latter, explicit through the sequencing construct (`let-in`), rather than fixed a-priori. In our calculus, the computational effect is *divergence*, so the effectful expressions will be called *possibly diverging*, whereas those effect-free will be called *value expressions*[6]. Note that, as customary, possibly diverging expressions are defined on top of value expressions, whereas the converse does not hold; such stratification will allow modularity in the technical development, as will be detailed in the following.

The fine-grained syntax is shown in Fig. 4. As said above, there are two distinct syntactic categories of values and possibly diverging expressions. For simplicity we use the same metavariable e of the surface syntax for the latter, though the defining production is changed. This is justified by the well-known fact that expressions of the surface language can be encoded in the fine-grained syntax, by using the `let-in` construct, and the injection of value expressions into expressions made explicit by the `return` keyword.

The resource-aware semantics is formally defined in Fig. 5. Corresponding to the two syntactic categories, such semantics is expressed by two distinct judgments, $v|\rho \Rightarrow_r \mathbf{v}|\rho'$ in the top section,

[6]They are often called just "values" in literature, though, as already noted in [Levy et al. 2003], they are not values in the operational sense, that is, results of the evaluation; here we keep the two notions distinct. Values turn out to be value expressions with no free variables, except that under a lambda.

$$\text{(VAR)}\ \frac{}{x|\rho, x : (s, \mathbf{v}) \Rightarrow_r \mathbf{v}|\rho, x : (s', \mathbf{v})}\ \begin{array}{l} r \preceq r' \\ r' + s' \preceq s \end{array} \qquad \text{(FUN)}\ \frac{}{\mathtt{rec}\, f.\lambda x.e|\rho \Rightarrow_r \mathtt{rec}\, f.\lambda x.e|\rho}$$

$$\text{(UNIT)}\ \frac{}{\mathtt{unit}|\rho \Rightarrow_r \mathtt{unit}|\rho} \qquad \text{(PAIR)}\ \frac{v_1|\rho \Rightarrow_{r\cdot r_1} \mathbf{v}_1|\rho_1 \quad v_2|\rho_1 \Rightarrow_{r\cdot r_2} \mathbf{v}_2|\rho_2}{\langle {}^{r_1} v_1, v_2 {}^{r_2}\rangle|\rho \Rightarrow_r \langle {}^{r_1}\mathbf{v}_1, \mathbf{v}_2 {}^{r_2}\rangle|\rho_2}$$

$$\text{(INL)}\ \frac{v|\rho \Rightarrow_{r\cdot s} \mathbf{v}|\rho'}{\mathtt{inl}^s v|\rho \Rightarrow_r \mathtt{inl}^s \mathbf{v}|\rho'} \qquad \text{(INL)}\ \frac{v|\rho \Rightarrow_{r\cdot s} \mathbf{v}|\rho'}{\mathtt{inr}^s v|\rho \Rightarrow_r \mathtt{inr}^s \mathbf{v}|\rho'}$$

$$\text{(RET)}\ \frac{v|\rho \Rightarrow_s \mathbf{v}|\rho'}{\mathtt{return}\ v|\rho \Rightarrow_r \mathbf{v}|\rho'}\ r \preceq s \neq \mathbf{0} \qquad \text{(LET)}\ \frac{\begin{array}{l} e_1|\rho \Rightarrow_s \mathbf{v}|\rho'' \\ e_2[x'/x]|\rho'', x' : (s, \mathbf{v}) \Rightarrow_r \mathbf{v}'|\rho' \end{array}}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2|\rho \Rightarrow_r \mathbf{v}'|\rho'}\ x' \text{ fresh}$$

$$\text{(APP)}\ \frac{\begin{array}{l} v_1|\rho \Rightarrow_s \mathtt{rec}\, f.\lambda x.e|\rho_1 \quad v_2|\rho_1 \Rightarrow_t \mathbf{v}_2|\rho_2 \\ e[f'/f][x'/x]|\rho_2, f' : (s_2, \mathtt{rec}\, f.\lambda x.e), x' : (t, \mathbf{v}_2) \Rightarrow_r \mathbf{v}|\rho' \end{array}}{v_1 v_2|\rho \Rightarrow_r \mathbf{v}|\rho'}\ \begin{array}{l} s_1 + s_2 \preceq s \\ s_1 \neq \mathbf{0} \\ f', x' \text{ fresh} \end{array}$$

$$\text{(MATCH-U)}\ \frac{v|\rho \Rightarrow_s \mathtt{unit}|\rho'' \quad e|\rho'' \Rightarrow_r \mathbf{v}|\rho'}{\mathtt{match}\ v\ \mathtt{with}\ \mathtt{unit}\ \to e|\rho \Rightarrow_r \mathbf{v}|\rho}\ s \neq \mathbf{0}$$

$$\text{(MATCH-P)}\ \frac{\begin{array}{l} v|\rho \Rightarrow_s \langle {}^{r_1}\mathbf{v}_1, \mathbf{v}_2 {}^{r_2}\rangle|\rho'' \\ e[x'/x][y'/y]|\rho'', x' : (s\cdot r_1, \mathbf{v}_1), y' : (s\cdot r_2, \mathbf{v}_2) \Rightarrow_r \mathbf{v}|\rho' \end{array}}{\mathtt{match}\ v\ \mathtt{with}\ \langle x, y\rangle \to e|\rho \Rightarrow_r \mathbf{v}|\rho'}\ \begin{array}{l} s \neq \mathbf{0} \\ x', y' \text{ fresh} \end{array}$$

$$\text{(MATCH-L)}\ \frac{\begin{array}{l} v|\rho \Rightarrow_t \mathtt{inl}^s \mathbf{v}|\rho'' \\ e_1[y/x_1]|\rho'', y : (t\cdot s, \mathbf{v}) \Rightarrow_r \mathbf{v}'|\rho' \end{array}}{\mathtt{match}\ v\ \mathtt{with}\ \mathtt{inl}\ x_1 \to e_1\ \mathtt{or}\ \mathtt{inr}\ x_2 \to e_2|\rho \Rightarrow_r \mathbf{v}'|\rho'}\ \begin{array}{l} t \neq \mathbf{0} \\ y \text{ fresh} \end{array}$$

$$\text{(MATCH-R)}\ \frac{\begin{array}{l} v|\rho \Rightarrow_t \mathtt{inr}^s \mathbf{v}|\rho'' \\ e_2[y/x_1]|\rho'', y : (t\cdot s, \mathbf{v}) \Rightarrow_r \mathbf{v}'|\rho' \end{array}}{\mathtt{match}\ v\ \mathtt{with}\ \mathtt{inl}\ x_1 \to e_1\ \mathtt{or}\ \mathtt{inr}\ x_2 \to e_2|\rho \Rightarrow_r \mathbf{v}'|\rho'}\ \begin{array}{l} t \neq \mathbf{0} \\ y \text{ fresh} \end{array}$$

Fig. 5. Resource-aware semantics

and $c \Rightarrow_r \mathbf{v}|\rho$ in the bottom section, with the latter defined on top of the former. Hence, the metarules in Fig. 5 can be equivalently seen as

- a unique inference system defining the union of the two judgments
- an inference system in the top section, defining $v|\rho \Rightarrow_r \mathbf{v}|\rho'$, and an inference system in the bottom section, defining $c \Rightarrow_r \mathbf{v}|\rho$, where the previous judgment acts as a side condition.

For simplicity, we use the same notation for the two judgments, and in the bottom section of Fig. 5 we write both judgments as premises, taking the first view. However, the second view will be useful later to allow a modular technical development.

Rules for value expressions just replace variables by values; such reduction cannot diverge, but is resource-consuming, hence can get stuck.

In particular, in (VAR), which is the key rule where resources are consumed, a variable is replaced by its associated value, and its grade s decreases to s', burning an amount r' of resource which has to be at least the reduction grade. The side condition $r' + s' \preceq s$ ensures that the initial grade allows to consume the r' amount, leaving a residual grade s'. Note that the consumed amount is *not* required to be exactly r, that is, there is no constraint that the semantics should not "waste* resources. Hence, reduction is largely non-deterministic; it will be the responsibility of the type system to ensure that there is at least one reduction which does not get stuck.

The other rules for value expressions propagate rule (VAR) to subterms which are variables. In rules for "data containers" (PAIR), (IN-L), and (IN-R), the components are evaluated with the evaluation grade of the compound value expression, multiplied by that of the component.

Whereas evaluation of value expressions may have grade $\mathbf{0}$, when they are actually *used*, that is, are subterms of possibly diverging expressions, they should be evaluated with a non-zero grade, as required by a side condition in the corresponding rules in the bottom section of Fig. 5.

In rule (RET), the evaluation grade of the value expression should be enough to cover the current evaluation grade. In rule (LET), expressions e_1 and e_2 are evaluated sequentially, the latter in an environment where the local variable x has been added as available resource, modulo renaming with a fresh variable to avoid clashes, with the value and grade obtained by the evaluation of e_1.

In rule (APP), an application $v_1\, v_2$ is evaluated by first consuming the resources needed to obtain a value from v_1 and v_2, with the former expected to be a (possibly recursive) function. Then, the function body is evaluated in an environment where the function name and the parameter have been added as available resources, modulo renaming with fresh variables. The function should be produced in a "number of copies", that is, with a grade s, enough to cover both all the future recursive calls (s_2) and the current use (s_1); in particular, for a non-recursive call, s_2 could be $\mathbf{0}$. Instead, the current use should be non-zero since we are actually using the function.

Note that s_1 is arbitrary, and could not be replaced by a sound default grade: notably, $\mathbf{1}$ would not work for grade algebras where there are grades between $\mathbf{0}$ and $\mathbf{1}$, as happens, e.g., for privacy levels.

Note also that, in this rule as in others, there is no required relation between the reduction grades of some premises (in this case, s and t) and that of the consequence, here r. Of course, depending on the choice of such grades, reduction could either proceed or get stuck due to resource exhaustion; the role of the type system is exactly to show that there is a choice which prevents the latter case.

Rules for match constructs, namely (MATCH-U), (MATCH-P), (MATCH-L), and (MATCH-R), all follow the same pattern. The resources needed to obtain a value from the value expression to be matched are consumed, and then the continuation is evaluated. In rule (MATCH-U), there is no value-passing from the matching expression to the continuation, hence their evaluation grades are independent. In rule (MATCH-P), instead, the continuation is evaluated in an environment where the two variables in the pattern have been added as available resources, again modulo renaming. The values associated to the two variables are that of the corresponding component of the expression to be matched, whereas the grades are the evaluation grade of such expression, multiplied by the grade of the component. Rules (MATCH-L) and (MATCH-R) are analogous.

Besides the standard typing errors, evaluation graded r can get stuck (formally, no judgment can be derived) when rule (VAR) cannot be applied since the side conditions do not hold. Informally, some resource (variable) is exhausted, that is, can no longer be replaced by its value. Also note that the instrumented reduction is non-deterministic, due to rule (VAR). That is, when a resource is needed, it can be consumed in different ways; hence, soundness of the type system will be *soundness-may*, meaning that *there exists* a computation which does not get stuck.

We end this section by illustrating resource consumption in a non-terminating computation.

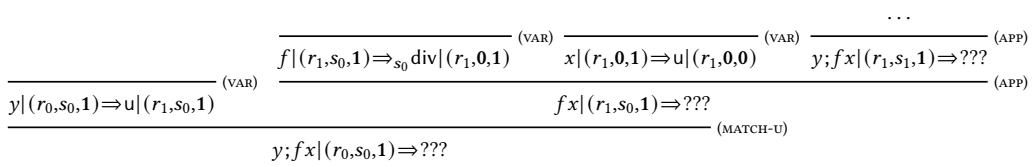

Fig. 6. Example of resource-aware evaluation: consumption/divergency

Example 3.3. Consider the function $\mathsf{div} = \mathtt{rec}\, f.\lambda x.y;fx$, which clearly diverges on any argument, by using infinitely many times the external resource y. In Fig. 6 we show the (incomplete) proof tree for the evaluation of an application $y;fx$, in an environment where f denotes the function div, and y and x denote `unit`. For simplicity, as in previous examples, we omit $\mathbf{1}$ grades, and renaming of variables; moreover, we abbreviate by (r,s,t) the environment $y : (r, \mathtt{unit}), f : (s, \mathsf{div}), x : (t, \mathtt{unit})$.

In this way, we can focus on the key feature the (tentative) proof tree shows: the body of div is evaluated infinitely many times, in a sequence of environments, starting from the root, where the grades of the external resource y, assuming that each time it is consumed by $\mathbf{1}$, are as follows:

$$r_0 = r_1 + \mathbf{1} \qquad r_1 = r_2 + \mathbf{1} \qquad \ldots \qquad r_k = r_{k+1} + \mathbf{1} \qquad \ldots$$

In the case of the resource f, at each recursive call, the function must be produced with a grade which is the sum of its current usage (assumed again to be $\mathbf{1}$) and the grade which will be associated to (a fresh copy of) f in the environment, to evaluate the body. As a consequence, we also get:

$$s_0 = s_1 + \mathbf{1} \qquad s_1 = s_2 + \mathbf{1} \qquad \ldots \qquad s_k = s_{k+1} + \mathbf{1} \qquad \ldots$$

Let us now see what happens depending on the underlying grade algebra, considering y (an analogous reasoning applies to f). Taking the grade algebra of natural numbers of Example 2.3(1), it is easy to see that the above sequence of constraints can be equivalently expressed as:

$$r_1 = r_0 - 1 \qquad r_2 = r_1 - 1 \qquad \ldots \qquad r_{k+1} = r_k - 1 \qquad \ldots$$

Thus, in a finite number of steps, the grade of y in the environment becomes 0, hence the proof tree cannot be continued since we can no longer extract the associated value by rule (VAR). In other words, the computation is stuck due to resource consumption.

Assume now to take, instead, natural numbers extended with ∞, as defined in Example 2.3(7). In this case, if we start with $r_0 = \infty$, intuitively meaning that y can be used infinitely many times, evaluation can proceed forever by taking $r_k = \infty$ for all k. The same happens if we take a non-quantitative grade algebra, e.g., that of privacy levels; we can have $r_k = \text{public}$ for all k. However, interpreting the rules in Fig. 5 in the standard inductive way, the semantics we get *does not* formalize such non-terminating evaluation, since we only consider judgments with a *finite* proof tree. We will see in Section 5 how to extend the semantics to model non-terminating computations as well.

4 RESOURCE-AWARE TYPE SYSTEM

Types, defined in Fig. 7, are those expected for the constructs in the syntax: functional, `Unit`, (tensor) product, sum, and (equi-)recursive types, obtained by interpreting the productions *coinductively*, so that infinite[7] terms are allowed. However, they are *graded*, that is, decorated with a grade, and the type subterms are graded. Moreover, accordingly with the fact that functions are possibly recursive, arrows in functional types are decorated with a grade as well, called *recursion grade* in the following, expressing the recursive usage of the function; thus, functional types decorated with $\mathbf{0}$ are non-recursive.

[7] More precisely, *regular* terms, that is, those with finitely many distinct subterms.

$$
\begin{array}{lcll}
\tau, \sigma & ::= & T \to_s S \mid \mathtt{Unit} \mid T \otimes S \mid T + S & \text{non-graded type} \\
T, S & ::= & \tau^r & \text{graded type} \\
\Gamma, \Delta & ::= & x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n & \text{(type-and-coeffect) context}
\end{array}
$$

Fig. 7. Types and (type-and-coeffect) contexts

In (type-and-coeffect) contexts, also defined in Fig. 7, order is immaterial and $x_i \neq x_j$ for $i \neq j$; hence, they represent maps from variables to pairs consisting of a grade (called *coeffect* when used in this position), and a (non-graded) type. Equivalently, a type-and-coeffect context can be seen as a pair consisting of the standard type context $x_1 : \tau_1 \ldots, x_n : \tau_n$, and the coeffect context $x_1 : r_1, \ldots, x_n : r_n$. We write $\mathsf{dom}(\Gamma)$ for $\{x_1, \ldots, x_n\}$.

We define the following operations on contexts:

- a partial order $\preceq$

$$
\begin{array}{rll}
\emptyset \preceq \emptyset & & \\
x :_s T, \Gamma \preceq x :_r T, \Delta & & \text{if } s \preceq r \text{ and } \Gamma \preceq \Delta \\
\Gamma \preceq x :_r T, \Delta & & \text{if } x \notin \mathsf{dom}(\Gamma) \text{ and } \Gamma \preceq \Delta \text{ and } \mathbf{0} \preceq r
\end{array}
$$

- a sum $+$

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

- a scalar multiplication $\cdot$

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

These operations on type-and-coeffect contexts are obtained by lifting the corresponding operations on coeffect contexts, which are the pointwise extension of those on coeffects (grades), 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.

In Fig. 8, we give the typing rules, which are *parameterized* on the underlying grade algebra. As for instrumented reduction, the resource-aware type system is formalized by two judgments, $\Gamma \vdash v : T$ and $\Gamma \vdash e : T$, for values and possibly diverging expressions, respectively. However, differently from reduction, the two judgments are mutually recursive, due to rule (T-FUN), hence the metarules in Fig. 8 define a unique judgment which is their union. We only comment the most significant points. In rule (T-SUB-V) and (T-SUB), the context can be made more general, and the grade of the type more specific. This means that, on one hand, variables can get less constraining coeffects. For instance, assuming affinity coeffects as in Example 2.3(2), 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, 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 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 variable is used. Here, more generally, the variable can get an arbitrary grade r, provided that the context is multiplied by r. The same "local promotion" can be applied in the following rules in the top section.

In rule (T-FUN), a (possibly recursive) function can get a graded functional type, provided that its body can get the result type in the context enriched by assigning the functional type to the function name, and the parameter type to the parameter. As mentioned, we expect the recursion

$$\text{(T-SUB-V)}\ \frac{\Gamma \vdash v : \tau^r \quad s \preceq r}{\Delta \vdash v : \tau^s \quad \Gamma \preceq \Delta} \qquad \text{(T-VAR)}\ \frac{}{x :_r \tau \vdash x : \tau^r}$$

$$\text{(T-FUN)}\ \frac{\Gamma, f :_s {\tau_1}^{r_1} \to_s {\tau_2}^{r_2}, x :_{r_1} \tau_1 \vdash e : {\tau_2}^{r_2}}{r{\cdot}\Gamma \vdash \mathtt{rec}\, f.\lambda x.e : ({\tau_1}^{r_1} \to_s {\tau_2}^{r_2})^r} \qquad \text{(T-UNIT)}\ \frac{}{\emptyset \vdash \mathtt{unit} : \mathtt{Unit}^r}$$

$$\text{(T-PAIR)}\ \frac{\Gamma_1 \vdash v_1 : {\tau_1}^{r_1} \quad \Gamma_2 \vdash v_2 : {\tau_2}^{r_2}}{r{\cdot}(\Gamma_1 + \Gamma_2) \vdash \langle {}^{r_1}v_1, {v_2}^{r_2}\rangle : ({\tau_1}^{r_1} \otimes {\tau_2}^{r_2})^r}$$

$$\text{(T-INL)}\ \frac{\Gamma \vdash v : \tau_1^{r_1}}{r{\cdot}\Gamma \vdash \mathtt{inl}^{r_1} v : (\tau_1^{r_1} + \tau_2^{r_2})^r} \qquad \text{(T-INR)}\ \frac{\Gamma \vdash v : \tau_2^{r_2}}{r{\cdot}\Gamma \vdash \mathtt{inr}^{r_2} v : (\tau_1^{r_1} + \tau_2^{r_2})^r}$$

$$\text{(T-SUB)}\ \frac{\Gamma \vdash e : \tau^r \quad s \preceq r}{\Delta \vdash e : \tau^s \quad \Gamma \preceq \Delta} \qquad \text{(T-RET)}\ \frac{\Gamma \vdash v : \tau^r}{\Gamma \vdash \mathtt{return}\ v : \tau^r}\ r \neq \mathbf{0}$$

$$\text{(T-LET)}\ \frac{\Gamma_1 \vdash e_1 : {\tau_1}^{r_1} \quad \Gamma_2, x :_{r_1} \tau_1 \vdash e_2 : {\tau_2}^{r_2}}{\Gamma_1 + \Gamma_2 \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : {\tau_2}^{r_2}} \qquad \text{(T-APP)}\ \frac{\Gamma_1 \vdash v_1 : ({\tau_1}^{r_1} \to_s {\tau_2}^{r_2})^{(r+r\cdot s)} \quad \Gamma_2 \vdash v_2 : {\tau_1}^{r\cdot r_1}}{\Gamma_1 + \Gamma_2 \vdash v_1 v_2 : {\tau_2}^{r\cdot r_2}}\ r \neq \mathbf{0}$$

$$\text{(T-MATCH-U)}\ \frac{\Gamma_1 \vdash v : \mathtt{Unit}^r \quad \Gamma_2 \vdash e : T}{\Gamma_1 + \Gamma_2 \vdash \mathtt{match}\ v\ \mathtt{with}\ \mathtt{unit} \to e : T}\ r \neq \mathbf{0}$$

$$\text{(T-MATCH-P)}\ \frac{\Gamma_1 \vdash v : (\tau_1^{r_1} \otimes \tau_2^{r_2})^r \quad \Gamma_2, x :_{r\cdot r_1} \tau, y :_{r\cdot r_2} \tau_2 \vdash e : T}{\Gamma_1 + \Gamma_2 \vdash \mathtt{match}\ v\ \mathtt{with}\ \langle x, y\rangle \to e : T}\ r \neq \mathbf{0}$$

$$\text{(T-MATCH-IN)}\ \frac{\Gamma_1 \vdash v : (\tau_1^{r_1} + \tau_2^{r_2})^r \quad \Gamma_2, x :_{r\cdot r_1} \tau_1 \vdash e_1 : T \quad \Gamma_2, x :_{r\cdot r_2} \tau_2 \vdash e_2 : T}{\Gamma_1 + \Gamma_2 \vdash \mathtt{match}\ v\ \mathtt{with}\ \mathtt{inl}\ x \to e_1\ \mathtt{or}\ \mathtt{inr}\ x \to e_2 : T}\ r \neq \mathbf{0}$$

Fig. 8. Typing rules

grade s to be $\mathbf{0}$ for a non-recursive function; for a recursive function, we expect s to be an "infinite" grade, in a sense which will be clarified in Example 4.1 below.

In the rules in the bottom section, when a value expression is used as subterm of a possibly diverging expression, its grade is required to be non-zero, since it is evaluated in the computation, hence its resource consumption should be taken into account.

In rule (T-APP), the function should be produced with a grade which is the sum of that required for the current usage (r) and that corresponding to the recursive calls: the latter are the grade required for a single usage multiplied by the recursion grade (s). For a non-recursive function ($s = \mathbf{0}$), the rule turns out to be as expected for a usual application.

Example 4.1. As an example of type derivation, we consider the function $\mathsf{div} = \mathtt{rec}\, f.\lambda x.y; f\, x$ introduced in Example 3.3. In Fig. 9, we show a (parametric) proof tree deriving for div a type of the shape $(\mathtt{Unit}^{r_1} \to_s \mathtt{Unit}^{r_2})$, in a context providing the external resource y. Consider, first of all, the condition that the recursion grade s should satisfy, that is $(r + r{\cdot}s) \preceq s$, for some $r \neq \mathbf{0}$, meaning that it should be enough to cover the recursive call in the body and all the further recursive calls.[8]. Assuming the grade algebra of natural numbers of Example 2.3(1), there is no grade s satisfying

[8]Note that r is arbitrary, since there is no sound default grade, and only required to be non-zero since the function is actually used.

$$\dfrac{\dfrac{}{y :_{r'} \mathsf{U} \vdash y : \mathsf{U}^{r'}}\,(\text{T-VAR}) \quad \dfrac{\dfrac{}{f :_{r+r\cdot s} \tau \vdash f : \tau^{r+r\cdot s}}\,(\text{T-VAR}) \quad \dfrac{}{x :_{r\cdot r_1} \mathsf{U} \vdash x : \mathsf{U}^{r\cdot r_1}}\,(\text{T-VAR})}{f :_{r+r\cdot s} \tau, x :_{r\cdot r_1} \mathsf{U} \vdash fx : \mathsf{U}^{r\cdot r_2}}\; r \neq \mathbf{0}\;(\text{T-APP})}{y :_{r'} \mathsf{U}, f :_{r+r\cdot s} \tau, x :_{r\cdot r_1} \mathsf{U} \vdash y; fx : \mathsf{U}^{r\cdot r_2}}\; r' \neq \mathbf{0}\;(\text{T-MATCH-U})$$

$$\dfrac{y :_{r'} \mathsf{U}, f :_{s} \tau, x :_{r_1} \mathsf{U} \vdash y; fx : \mathsf{U}^{r_2}}{y :_{r'} \mathsf{U} \vdash \mathtt{rec}\, f.\lambda x.y; fx : \tau}\,(\text{T-FUN}) \qquad \begin{array}{l}(r + r\cdot s) \preceq s \quad r\cdot r_1 \preceq r_1 \quad r_2 \preceq r\cdot r_2\\ \tau = \mathsf{U}^{r_1} \to_s \mathsf{U}^{r_2}\end{array}\;(\text{T-SUB})$$

Fig. 9. Example of type derivation: recursive function

this condition. In other words, the type system correctly rejects the function since its application would get stuck due to resource consumption, as illustrated in Example 3.3. On the other hand, for, e.g., $s = \infty$, with natural numbers extended as in Example 2.3(7), $(r + r\cdot s) \preceq s$ would hold for any $r \neq 0$, hence the function would be well-typed. Moreover, there would be no constraints on the parameter and return type, since the conditions $r\cdot r_1 \preceq r_1$ and $r_2 \preceq r\cdot r_2$ would be always satified taking $r = 1$. Assuming the grade algebra of privacy levels introduced before Example 3.2, where $\mathbf{1}$ = public, for s = public the condition is satisfied analogously, again with no constraints on r_1 and r_2. For s = private, instead, it only holds for r = private, hence the condition $r_2 \preceq r\cdot r_2$ prevents the return type of the function to be public, accordingly with the intuition that a function used in private mode cannot return a public result. In such cases, the type system correctly accepts the function since its application to a value never gets stuck. Finally note that, to type an application of the function, e.g., to derive that div u has type U^{r_2}, div should get grade $\mathbf{1} + s$, hence the grade of the external resource y should be $(\mathbf{1} + s)\cdot r'$, that is, it should be usable infinitely many times as well.

A similar reasoning applies in general; namely, for recursive calls in a function's body we get a condition of shape $(r + r\cdot s) \preceq s$, with $r \neq \mathbf{0}$, forcing the grade s of the function to be "infinite". This happens regardless the recursive function is actually always diverging, as in the example above, or terminating on some/all arguments. On the other hand, in the latter case the resource-aware semantics terminates, as expected, provided that the initial amount of resources is enough to cover the (finite number of) recursive calls. This is perfectly reasonable, the type system being a static overapproximation of the evaluation. We will show an example of this terminating resource-aware evaluation in Section 6 (Fig. 15).

The following lemmas show that the promotion rule, usually explicitly stated in graded type systems, is admissible in our system (Lemma 4.2), and also a converse holds for value expressions (Lemma 4.3). Note that we can promote an expression only using a non-zero grade, to ensure that non-zero constraints on grades in typing rules for expressions are preserved.[9] These lemmas also show that we can assign to a value expression any grade provided that the context is appropriately adjusted: by demotion we can always derive $\mathbf{1}$ (taking $r = \mathbf{1}$) and then by promotion any grade.

Lemma 4.2 (Promotion).
(1) *If* $\Gamma \vdash v : \tau^r$ *then* $s\cdot\Gamma \vdash v : \tau^{s\cdot r}$
(2) *If* $\Gamma \vdash e : \tau^r$ *and* $s \neq \mathbf{0}$, *then* $s\cdot\Gamma \vdash e : \tau^{s\cdot r}$.

Lemma 4.3 (Demotion). *If* $\Gamma \vdash v : \tau^{s\cdot r}$ *then* $s\cdot\Gamma' \preceq \Gamma$ *and* $\Gamma' \vdash v : \tau^r$, *for some* Γ' .

In Fig. 10 we give the typing rules for environments and configurations. In such rules, Γ is the context whose domain is that of the environment, and each variable has as coeffect its grade in the environment, and as type that of its value. The side conditions use the relation $\preceq_\bullet$ defined as follows:

[9]Without assuming the grade algebra to be integral, we would need to use grades which are not zero-divisors.

$$\text{(T-ENV)}\ \frac{\Gamma_i \vdash \mathbf{v}_i : \tau_i^{\mathbf{1}} \quad \forall i \in 1..n}{\Gamma \vdash \rho \triangleright r_1 \cdot \Gamma_1 + \ldots + r_n \cdot \Gamma_n} \begin{array}{l} \rho = x_1 : (r_1, \mathbf{v}_1), \ldots, x_n : (r_n, \mathbf{v}_n) \\ \Gamma = x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \\ (r_1 \cdot \Gamma_1 + \ldots + r_n \cdot \Gamma_n) \preceq_\bullet \Gamma \end{array}$$

$$\text{(T-VCONF)}\ \frac{\Delta \vdash v : T \qquad \Gamma \vdash \rho \triangleright \hat{\Gamma}}{\Gamma \vdash v|\rho : T}\ \hat{\Gamma} + \Delta \preceq_\bullet \Gamma \qquad \text{(T-CONF)}\ \frac{\Delta \vdash e : T \qquad \Gamma \vdash \rho \triangleright \hat{\Gamma}}{\Gamma \vdash e|\rho : T}\ \hat{\Gamma} + \Delta \preceq_\bullet \Gamma$$

Fig. 10. Typing rules for environments and configurations

$\Delta \preceq_\Theta \Gamma$ if $\Delta + \Theta \preceq \Gamma$
$\Delta \preceq_\bullet \Gamma$ if $\Delta \preceq_\Theta \Gamma$ for some Θ

In rule (T-ENV), the side condition requires coeffects (grades) of variables in the environment to be enough to cover their uses in all the corresponding values; in rules (T-VCONF) and (T-CONF) to be enough to cover their uses in the expression as well. In the relation $\Delta \preceq_\Theta \Gamma$, the context Θ, called *residual context* in the following, is needed since variables in the environment may have an arbitrary grade, whereas, in the relation $\Delta \preceq \Gamma$, grades of variables in Γ should overapproximate those in Δ. For instance, taking the linearity grade algebra of Example 2.3(2), Γ could not add linear variables which are unused in both the expression and the codomain of the environment. In other words, the residual context allows resources to be, in a sense, "wasted", accordingly with the instrumented semantics, where there is no check that available resources are fully consumed. This could be refined at the price of a more involved semantics.

5 TYPE SOUNDNESS

In this section, we prove our main result: soundness of the type system with respect to the resource-aware big-step semantics. That is, for well-typed expressions there is a computation which is not stuck for any reason, including resource consumption. Note that this is a *may* flavour of soundness [Dagnino 2022; Dagnino et al. 2020; De Nicola and Hennessy 1984], which is the only one we can prove in this context, because resource consumption is non-deterministic, thus one can always get stuck consuming more resources than needed. We analyse separately type soundness for value and possibly diverging expressions.

Type soundness for value expressions. Since reduction of value expressions cannot diverge, type soundness means that, if well-typed, then they reduce to a value, as stated below.

Theorem 5.1 (Soundness for value expressions). *If* $\Gamma \vdash v|\rho : \tau^r$, *then* $\vdash v|\rho \Rightarrow_r \mathbf{v}|\rho'$ *for some* $\mathbf{v}$, ρ'.

We derive this theorem as a corollary of the following lemma, stating that, if a value expression and environment are well-typed with a given residual context, then they reduce to a value and environment which are well-typed with the same residual context.

Lemma 5.2 (Progress/Type preservation for value expressions). *If* $\Delta \vdash v : \tau^r$ *and* $\Gamma \vdash \rho \triangleright \hat{\Gamma}$ *with* $\hat{\Gamma} + \Delta \preceq_\Theta \Gamma$ *then* $v|\rho \Rightarrow_r \mathbf{v}|\rho'$ *and* $\Delta' \vdash \mathbf{v} : \tau^r$ *and* $\Gamma' \vdash \rho' \triangleright \hat{\Gamma}'$ *with* $\hat{\Gamma}' + \Delta' \preceq_\Theta \Gamma'$.

The proof of this result is by induction on the structure of values, relying on standard lemmas. Note that, even though reduction of value expressions just performs substitution, in a resource-aware semantics this is a significant event, since it implies consuming some amount of resources. The lemma above states that no resource exhaustion can happen, playing the role of progress plus type preservation (subject reduction) in small-step semantics. However, rather than saying that reduction cannot get stuck, since it is non-diverging we can simply say that there is a final result.

$$\mathrm{R} \quad ::= \quad \mathbf{v}|\rho \mid \infty \qquad \text{result}$$

$$(\textsc{ret})\ \frac{v|\rho \Rightarrow_s \mathbf{v}|\rho'}{\mathtt{return}\ v|\rho \Rightarrow_r \mathbf{v}|\rho'}\ r \preceq s \neq \mathbf{0} \qquad (\textsc{let-div1})\ \frac{e_1|\rho \Rightarrow_s \infty}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2|\rho \Rightarrow_r \infty}$$

$$(\textsc{let/let-div2})\ \frac{\begin{array}{l} e_1|\rho \Rightarrow_s \mathbf{v}|\rho'' \\ e_2[x'/x]|\rho'', x' : (s, \mathbf{v}) \Rightarrow_r \boxed{\mathrm{R}} \end{array}}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2|\rho \Rightarrow_r \boxed{\mathrm{R}}}\ x' \text{ fresh}$$

$$(\textsc{app/app-div})\ \frac{\begin{array}{l} v_1|\rho \Rightarrow_s \mathtt{rec}\, f.\lambda x.e|\rho_1 \quad v_2|\rho_1 \Rightarrow_t \mathbf{v}_2|\rho_2 \\ e[x'/x][f'/f]|\rho_2, x' : (t, \mathbf{v}_2), f' : (s_2, \mathtt{rec}\, f.\lambda x.e) \Rightarrow_r \boxed{\mathrm{R}} \end{array}}{v_1\, v_2|\rho \Rightarrow_r \boxed{\mathrm{R}}}\ \begin{array}{l} x', f' \text{ fresh} \\ s_1 + s_2 \preceq s \\ s_1 \neq \mathbf{0} \end{array}$$

$$(\textsc{match-p/match-p-div})\ \frac{\begin{array}{l} v|\rho \Rightarrow_s \langle {}^{r_1}\mathbf{v}_1, \mathbf{v}_2{}^{r_2}\rangle|\rho'' \\ e[x'/x][y'/y]|\rho'', x' : (s{\cdot}r_1, \mathbf{v}_1), y' : (s{\cdot}r_2, \mathbf{v}_2) \Rightarrow_r \boxed{\mathrm{R}} \end{array}}{\mathtt{match}\ v\ \mathtt{with}\ \langle x, y\rangle \to e|\rho \Rightarrow_r \boxed{\mathrm{R}}}\ x', y' \text{ fresh}$$

$$(\textsc{match-l/match-l-div})\ \frac{v|\rho \Rightarrow_s \mathtt{inl}^r \mathbf{v}|\rho'' \quad e_1[y/x_1]|\rho'', y : (s{\cdot}r, \mathbf{v}) \Rightarrow_t \boxed{\mathrm{R}}}{\mathtt{match}\ v\ \mathtt{with}\ \mathtt{inl}\, x_1 \to e_1\ \mathtt{or}\ \mathtt{inr}\, x_2 \to e_2|\rho \Rightarrow_t \boxed{\mathrm{R}}}\ y \text{ fresh}$$

$$(\textsc{match-r/match-r-div})\ \frac{v|\rho \Rightarrow_s \mathtt{inr}^r \mathbf{v}|\rho'' \quad e_2[y/x_2]|\rho', y : (s{\cdot}r, \mathbf{v}) \Rightarrow_t \boxed{\mathrm{R}}}{\mathtt{match}\ v\ \mathtt{with}\ \mathtt{inl}\, x_1 \to e_1\ \mathtt{or}\ \mathtt{inr}\, x_2 \to e_2|\rho \Rightarrow_t \boxed{\mathrm{R}}}\ y \text{ fresh}$$

$$(\textsc{match-u/match-u-div})\ \frac{v|\rho \Rightarrow_r \mathtt{unit}|\rho'' \quad e|\rho'' \Rightarrow_t \boxed{\mathrm{R}}}{\mathtt{match}\ v\ \mathtt{with}\ \mathtt{unit}\ \to e|\rho \Rightarrow_t \boxed{\mathrm{R}}}\ x' \text{ fresh} \qquad (\textsc{co-div})\ \frac{}{e|\rho \Rightarrow_r \infty}$$

Fig. 11. Adding divergence

Adding divergence. For possibly diverging expressions, instead, the big-step semantics defined in Fig. 5 suffers from the long-known drawback [Cousot and Cousot 1992; Leroy and Grall 2009] that non-terminating and stuck computations are indistinguishable, since in both cases no finite proof tree of a judgment can be constructed. This is an issue for our aim: to prove that for a well-typed expression there is a resource-aware evaluation which does not get stuck, that is, either produces a value or diverges. To solve this problem, we extend the big-step semantics to explicitly model diverging computations, proceeding as follows:

- the judgment for value expressions remains defined as in the top section of Fig. 5
- the shape of the judgment for possibly diverging expressions is generalized to $c \Rightarrow_r \mathrm{R}$, where the *result* R is either a pair consisting of a value and a final environment, or *divergence* (∞);
- this judgment is defined through a *generalized inference system*, shown in Fig. 11, consisting of the *rules* from (RET) to (MATCH-U/MATCH-U-DIV), and the *corule* (CO-DIV) (differences with respect to the previous semantics in the bottom section of Fig. 5 are emphasized in grey[10]).

The key point here is that, in generalized inference systems, rules are interpreted in an *essentially coinductive*, rather than inductive, way. For details on generalized inference systems we refer to [Ancona et al. 2017a; Dagnino 2019]; here, for the reader's convenience, we provide a self-contained presentation, instantiating general definitions on our specific case.

[10]Recall that, since the evaluation judgment is stratified, premises involving the judgment for value expressions can be equivalently considered as side conditions.

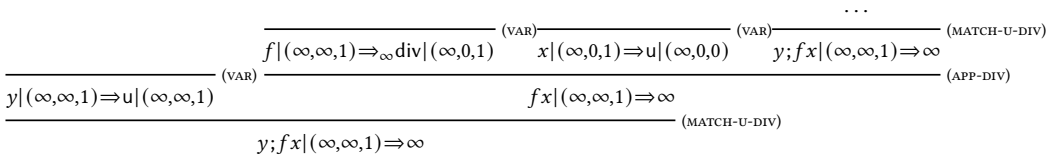

Fig. 12. Example of resource-aware evaluation: divergency with no consumption

Rules in Fig. 11 handle divergence propagation. Notably, for each rule in the bottom section of Fig. 5, we add a divergence propagation rule for each of the possibly diverging premises. The only rule with two possibly diverging premises is (LET). Hence, divergence propagation for an expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ is obtained by two (meta)rules: (LET-DIV1) when e_1 diverges, and (LET-DIV2) when e_1 converges and e_2 diverges; in Fig. 11, for brevity, this metarule is merged with (LET), using the metavariable R. All the other rules have only one possibly diverging premise, so one divergence propagation rule is added and merged with the original metarule, analogously to (LET-DIV2).

In generalized inference systems, infinite proof trees are allowed. Hence, judgments $c \Rightarrow_r \infty$ can be derived, as desired, even though there is no axiom introducing them, thus distinguishing diverging computations (infinite proof tree) from stuck computations (no proof tree). However, a purely coinductive interpretation would allow the derivation of spurious judgements [Ancona et al. 2017b; Cousot and Cousot 1992; Leroy and Grall 2009]. To address this issue, generalized inference systems may include *corules*, written with a thick line, only (CO-DIV) in our case, which refine the coinductive interpretation, filtering out some (undesired) infinite derivations. Intuitively, the meaning of (CO-DIV) is to allow infinite derivations only for divergence (see Example 5.4 below). Formally, we have the following definition instantiated from [Ancona et al. 2017a; Dagnino 2019].

Definition 5.3. A judgment $c \Rightarrow_r \mathrm{R}$ is derivable in the generalized inference system in Fig. 11, written $\vdash_\infty c \Rightarrow_r \mathrm{R}$, if it has an infinite proof tree constructed using the rules where, moreover, each node has a *finite* proof tree constructed using the rules *plus the corule*.

Example 5.4. Let us consider again the expression $y; f\,x$ of Example 3.3. Now, its non-terminating evaluation in the environment $y : (\infty, \texttt{unit}), f : (\infty, \mathrm{div}), x : (1, \texttt{unit})$, abbreviated $(\infty,\infty,1)$ using the previous convention, is formalized by the infinite proof tree in Fig. 12, where instantiations of (meta)rules (MATCH-U) and (APP) have been replaced by those of the corresponding divergence propagation rule. It is immediate to see that each node in such infinite proof tree has a finite proof tree constructed using the rules plus the corule: the only nodes which have no finite proof tree constructed using the rules are those in the infinite path, of shape either $y; f\,x|(\infty,\infty,1) \Rightarrow \infty$ or $f\,x|(\infty,\infty,1) \Rightarrow \infty$, and such judgments are directly derivable by the corule. On the other hand, infinite proof trees obtained by using (MATCH-U) and (APP) would derive $y; f\,x|(\infty,\infty,1) \Rightarrow \mathbf{v}|(\infty,\infty,1)$ for any $\mathbf{v}$. However, such judgments *have no* finite proof tree using also the corule, which allows only to introduce divergence, since there is no rule deriving a value with divergence as a premise.

The transformation from the inductive big-step semantics in Fig. 5 to that handling divergence in Fig. 11 is an instance of a general construction, taking as input an arbitrary big-step semantics, fully formalized, and proved to be correct, in [Dagnino 2022]. In particular, the construction is *conservative*, that is, the semantics of converging computations is not affected, as stated in the following result, which is an instance of Theorem 6.3 in [Dagnino 2022].

Definition 5.5. Let $\vdash c \Rightarrow_r \mathbf{v}|\rho$ denote that the judgment can be derived by the rules in the bottom section of Fig. 5, interpreted inductively.

THEOREM 5.6 (CONSERVATIVITY). $\vdash_\infty c \Rightarrow_r \mathbf{v}|\rho'$ *if and only if* $\vdash c \Rightarrow_r \mathbf{v}|\rho'$.

Note that to achieve this result corules are essential since, as observed in Example 5.4, a purely coinductive interpretation allows for infinite proof trees deriving values.

Type soundness for possibly diverging expressions. The definition of the semantics by the generalized inference system in Fig. 11 allows a very simple and clean formulation of type soundness: well-typed configurations always reduce to a result (which can be possibly divergence). Formally:

Theorem 5.7 (Soundness). *If* $\Gamma \vdash c : \tau^r$, *then* $\vdash_\infty c \Rightarrow_r R$ *for some* R.

We describe now the structure of the proof, which is interesting in itself; indeed, the semantics being big-step, there is no consolidated proof technique as the long-time known progress plus subject reduction for the small-step case [Wright and Felleisen 1994].

The proof is driven by coinductive reasoning on the semantic rules, following a schema firstly adopted in [Ancona et al. 2017b], as detailed below. First of all, to the aim of such proof, it is convenient to turn to the following equivalent formulation of type soundness, stating that well-typed configurations which do not converge necessarily diverge.

Theorem 5.8 (Completeness-∞). *If* $\Gamma \vdash c : \tau^r$, *and there is no* $\mathbf{v}|\rho$ *s.t.* $\vdash_\infty c \Rightarrow_r \mathbf{v}|\rho$, *then* $\vdash_\infty c \Rightarrow_r \infty$.

Indeed, with this formulation soundness of the type system can be seen as *completeness* of the set of judgements $c \Rightarrow_r \infty$ which are derivable with respect to the set of pairs (c, r) such that c is well-typed with grade r, and does not converge. The standard technique to prove completeness of a coinductive definition with respect to a specification S is the coinduction principle, that is, by showing that S is *consistent* with respect to the coinductive definition. This means that each element of S should be the consequence of a rule whose premises are in S as well. In our case, since the definition of $\vdash_\infty c \Rightarrow_r \infty$ is not purely coinductive, but refined by the corule, completeness needs to be proved by the *bounded coinduction* principle [Ancona et al. 2017a; Dagnino 2019], a generalization of the coinduction principle. Namely, besides proving that S is consistent, we have to prove that S is *bounded*, that is, each element of S can be derived by the inference system consisting of the rules *and the corules*, in our case, only (co-div), interpreted inductively.

The proof of Theorem 5.8 modularly relies on two results. The former (Theorem 5.9) is the instantiation of a general result proved in [Ancona et al. 2017b] (Theorem 3.3) by bounded coinduction. For the reader's convenience, to illustrate the proof technique in a self-contained way, we report here statement and proof for our specific case. Namely, Theorem 5.9 states completeness of diverging configurations with respect to any family of configurations which satisfies the *progress-∞ property*. The latter (Theorem 5.10) is the progress-∞ property for our type system.

Theorem 5.9 (Progress-$\infty \Rightarrow$ Completeness-∞). *For each grade* r, *let* C_r *be a set of configurations, and set* $C_r^\infty = \{c \in C_r \mid \nexists\, \mathbf{v}|\rho \text{ such that } \vdash c \Rightarrow_r \mathbf{v}|\rho\}$. *If the following condition holds:*[11]

> *(progress-∞)* $c \in C_r^\infty$ *implies that* $c \Rightarrow_r \infty$ *is the consequence of a rule where, for all premises of shape* $c' \Rightarrow_s \infty$, $c' \in C_s^\infty$, *and, for all premises of shape* $c' \Rightarrow_s R$, *with* $R \neq \infty$, $\vdash c' \Rightarrow_s R$.

then $c \in C_r^\infty$ *implies* $\vdash_\infty c \Rightarrow_r \infty$.

Proof. We set $S = \{c \Rightarrow_r \infty \mid c \in C_r^\infty\} \cup \{c \Rightarrow_r R \mid R \neq \infty, \vdash c \Rightarrow_r R\}$, and prove that, for each $c \Rightarrow_r R \in S$, we have $\vdash_\infty c \Rightarrow_r R$, by bounded coinduction. We have to prove two conditions.

(1) S is consistent, that is, each $c \Rightarrow_r R$ in S is the consequence of a rule whose premises are in S as well. We reason by cases:

[11] Keep in mind that $c \Rightarrow_r R$ denotes just the judgment (a triple), whereas $\vdash c \Rightarrow_r R$ and $\vdash_\infty c \Rightarrow_r R$ denote derivability of the judgment (Definition 5.5 and Definition 5.3, respectively).

- For each $c \Rightarrow_r \infty \in S$, by the (PROGRESS-$\infty$) hypothesis it is the consequence of a rule where, for all premises of shape $c' \Rightarrow_s \infty$, $c' \in C_s^\infty$, hence $c' \Rightarrow_s \infty \in S$, and, for all premises of shape $c' \Rightarrow_s \mathrm{R}$, with $\mathrm{R} \neq \infty$, $\vdash c' \Rightarrow_s \mathrm{R}$, hence $c' \Rightarrow_s \mathrm{R} \in S$ as well.
- For each $c \Rightarrow_r \mathbf{v}|\rho \in S$, we have $\vdash c \Rightarrow_r \mathbf{v}|\rho$, hence this judgment is the consequence of a rule in Fig. 5 where for each premise, necessarily of shape $c' \Rightarrow_{r'} \mathbf{v}'|\rho'$, we have $\vdash c' \Rightarrow_{r'} \mathbf{v}'|\rho'$, hence $c' \Rightarrow_{r'} \mathbf{v}'|\rho' \in S$.

(2) S is bounded, that is, each $c \Rightarrow_r \mathrm{R}$ in S can be inductively derived (has a finite proof tree) using the rules and the corule in Fig. 11. This is trivial, since, for $\mathrm{R} = \infty$, the judgment can be directly derived by (CO-DIV), and, for $\mathrm{R} \neq \infty$, since $\vdash c \Rightarrow_r \mathrm{R}$, this holds by definition.

□

Thanks to the theorem above, to prove type soundness (formulated as in Theorem 5.8) it is enough to prove the progress-∞ property for well-typed configurations which do not converge. The name is chosen to suggest the analogous of progress in small-step semantics, meaning that, for a non-converging well-typed configuration, the construction of a proof tree can never get stuck.

Set $WT_r = \{c \mid \Gamma \vdash c : \tau^r \text{ for some } \Gamma, \tau\}$, and, accordingly with the notation in Theorem 5.9, $WT_r^\infty = \{c \mid c \in WT^r \text{ and } \nexists\, \mathbf{v}|\rho \text{ such that } \vdash_\infty c \Rightarrow_r \mathbf{v}|\rho\}$.

THEOREM 5.10 (PROGRESS-∞). *If $c \in WT_r^\infty$, then $c \Rightarrow_r \infty$ is the consequence of a rule where, for all premises of shape $c' \Rightarrow_s \infty$, $c' \in WT_s^\infty$, and, for all premises of shape $c' \Rightarrow_s \mathrm{R}$, with $\mathrm{R} \neq \infty$, $\vdash c' \Rightarrow_s \mathrm{R}$.*

We derive this theorem as a corollary of the next lemma, which needs the following notations:

- We use the metavariable $\tilde{\rho}$ for environments where grades have been erased, hence they are maps from variables into values.
- We write $\mathrm{erase}(\rho)$ for the environment obtained from ρ by erasing grades.
- The reduction relation $\Rightarrow$ over pairs $v|\tilde{\rho}$ and $e|\tilde{\rho}$ is defined by the metarules in Fig. 5 where we remove side conditions involving grades. That is, such relation models standard semantics.

The lemma states that, if an expression and environment are well-typed with a given residual context, and (ignoring the grades) they reduce to a value and environment, then the value is well-typed, and the environment *can be*[12] decorated with grades to be well-typed, with the same residual context. Note that, differently from Lemma 5.2, here the hypothesis of well-typedness of the configuration is not enough, but we need also to assume progress of standard reduction.

LEMMA 5.11 (TYPE PRESERVATION). *If $\Delta \vdash e : \tau^r$ and $\Gamma \vdash \rho_1 \triangleright \hat{\Gamma}$ with $\hat{\Gamma} + \Delta \preceq_\Theta \Gamma$ and, set $\tilde{\rho}_1 = \mathrm{erase}(\rho_1)$, $e|\tilde{\rho}_1 \Rightarrow \mathbf{v}|\tilde{\rho}_2$, then $\Delta' \vdash \mathbf{v} : \tau^r$ and $\Gamma' \vdash \rho_2 \triangleright \hat{\Gamma}'$ with $\hat{\Gamma}' + \Delta' \preceq_\Theta \Gamma'$, for some ρ_2 such that $\mathrm{erase}(\rho_2) = \tilde{\rho}_2$.*

6 PROGRAMMING EXAMPLES AND DISCUSSIONS

In this section, for readability, we use the surface syntax and, moreover, assume some standard additional constructs and syntactic conventions. Notably, we generalize (tensor) product types to tuple types, with the obvious extended syntax, and sum types to variant types, written $\ell_1{:}T_1 + \ldots + \ell_n{:}T_n$ for some *tags* $\ell_1, \ldots, \ell_n$, injections generalized to tagged variants $\ell^r e$, and matching of the shape $\texttt{match}\ e\ \texttt{with}\ \ell_1 x_1\ \texttt{or}\ \ldots\ \texttt{or}\ \ell_n x_n$. We write just ℓ as an abbreviation for an addend $\ell{:}\texttt{Unit}^{\mathbf{0}}$ in a variant type, and also for the corresponding tagged variants $\ell^{\mathbf{0}}\texttt{unit}$ and patterns $\ell\, x$ in a matching construct. Moreover, we will use type and function definitions (that is, synonyms for function and type expressions), and, as customary, represent (equi-)recursive types by equations. Finally, we will omit **1** annotations as in the previous examples, and we will consider **0** as default, hence omitted, as recursion grade (that is, functions are by default non-recursive).

[12]That is, as soundness, type preservation holds in the *may* flavour.

```
Bool = true + false
Nat = zero + succ:Nat
NatList = empty + cons:(Nat ⊗ NatList)
OptNat = none + some:Nat

not: Bool -> Bool
not = \b.match b with true -> false or false -> true

even: Nat ->∞ Bool
even = rec ev.\n.match n with zero -> true or succ m -> not (ev m)

_+_: Nat ->∞ Nat -> Nat
_+_ = rec sum.\n.\m. match n with zero -> m or succ x -> succ (sum x m)

double: Nat^2 -> Nat
double n = n + n

_*_: Nat ->∞ Nat^∞ -> Nat
_*_ = rec mult.\n.\m.match n with zero -> zero or succ x -> (mult x m) + m

length : NatList ->∞ Nat
length = rec len.
         \ls.match ls with empty -> zero or cons ls1 -> match ls1 with <_,tl> -> succ (len tl)

get : NatList ->∞ Nat -> OptNat
get = rec g.
         \ls.\i.match ls with empty -> none
                or cons ls1 ->
                  match ls1 with <hd,tl> -> match i with zero -> some hd or succ j -> g j tl
```

Fig. 13. Examples of type and function definitions

Natural numbers and lists. The encoding of booleans, natural numbers, lists of natural numbers, and optional natural numbers, is given at the top of Fig. 13 followed by the definition of some standard functions. Assume, firstly, the grade algebra of natural numbers with bounded usage, Example 2.3(1), extended with ∞, as in Example 2.3(7), needed as annotation of recursive functions, as has been illustrated in Example 4.1; we discuss below what happens taking exact usage instead.

As a first comment, note that in types of recursive functions the recursion grade needs to be ∞, as expected. On the other hand, most function parameters are graded 1, since they are used at most once in each branch of the function's body. The second parameter of multiplication, instead, needs to be graded ∞. Indeed, it is used in the body of the function both as argument of sum, and *inside* the recursive call. Hence, its grade r should satisfy the equation $(\mathbf{1} + r) \leq r$, analogously to what happens for the recursive grade; compare with the parameter of function `double`, which is used twice as well, and can be graded 2. In the following alternative definitions:

```
double: Nat^1 -> Nat
double n = n * succ succ zero

double: Nat^∞ -> Nat
double n = succ succ zero * n
```

the parameter needs to be graded differently depending on how it is used in the multiplication. In other words, the resource-aware type system captures, as expected, non-extensional properties.

Assume now the grade algebra of natural numbers with exact usage, again extended with ∞. Interestingly enough, the functions `length` and `get` above are no longer typable.

$$\begin{array}{lll} v & ::= & \ldots \mid [^{r_1} v_1, v_2{}^{r_2}] \mid \pi_1 v \mid \pi_2 v \\ \tau, \sigma & ::= & \ldots \mid T \times S \end{array}$$

$$\text{(APAIR)}\ \frac{v_1|\rho \Rightarrow_{r\cdot r_1} \mathbf{v}_1|\rho' \quad v_2|\rho \Rightarrow_{r\cdot r_2} \mathbf{v}_2|\rho'}{[^{r_1} v_1, v_2{}^{r_2}]|\rho \Rightarrow_r [^{r_1}\mathbf{v}_1, \mathbf{v}_2{}^{r_2}]|\rho'} \qquad \text{(PROJ)}\ \frac{v|\rho \Rightarrow_s [^{r_1}\mathbf{v}_1, \mathbf{v}_2{}^{r_2}]|\rho'}{\pi_i v|\rho \Rightarrow_r \mathbf{v}_i|\rho'}\ \begin{array}{l} i \in \{1,2\} \\ r \preceq s\cdot r_i \end{array}$$

$$\text{(T-APAIR)}\ \frac{\Gamma \vdash v_1 : \tau_1{}^{r_1} \quad \Gamma \vdash v_2 : \tau_2{}^{r_2}}{r\cdot\Gamma \vdash [^{r_1} v_1, v_2{}^{r_2}] : (\tau_1{}^{r_1} \times \tau_2{}^{r_2})^r} \qquad \text{(T-PROJ)}\ \frac{\Gamma \vdash v : (\tau_1^{r_1} \times \tau_2^{r_2})^r}{\Gamma \vdash \pi_i v : \tau_i{}^{r\cdot r_i}}\ r \neq \mathbf{0}$$

Fig. 14. Cartesian product

In `length`, this is due to the fact that, when the list is non-empty, the head is unused, whereas, since the grade of a pair is "propagated" to both the components, it should be used exactly once as the tail. The function would be typable assuming for lists the type `NatList=empty+cons:(Nat`$^{\mathbf{0}}$`⊗NatList)`, which, however, would mean to essentially handle a list as a natural number.

Function `get`, analogously, cannot be typed since, in the last line, only one component of a non-empty list (either the head or the tail) is used in a branch of the match, whereas both should be used exactly once. Both functions could be typed, instead, grading with ∞ the list parameter; this would mean to allow an arbitrary usage in the body. These examples suggest that, in a grade algebra with exact usage, such as that of natural numbers, or the simpler linearity grade algebra, see Example 2.3(2), there is often no middle way between imposing severe limitations on code, to ensure linearity (or, in general, exactness), and allowing code which is essentially non-graded.

Additive product. The product type we consider in Fig. 7 is the *tensor* product, also called *multiplicative*, following Linear Logic terminology [Girard 1987]. In the destructor construct for such type, both components are simultaneously extracted, each one with a grade which is (a multiple of) that of the pair, see the semantic[13] rule (MATCH-P) in Fig. 5. Thus, as shown in the examples above, programs which discard the use of either component cannot be typed in a non-affine grade algebra. Correspondingly, the resource consumption for constructing a (multiplicative) pair is the *sum* of those for constructing the two components, corresponding to a sequential evaluation, see rule (PAIR) in Fig. 5. The *cartesian* product, instead, also called *additive*, formalized in Fig. 14, has one destructor for each component, so that which is not extracted is discarded. Correspondingly, the resource consumption for constructing an additive pair is *an upper bound* of those for constructing the two components, corresponding in a sense to a non-deterministic evaluation. The `get` example, rewritten using the constructs of the additive product, becomes typable even in a non-affine grade algebra. In an affine grade algebra, programs can always be rewritten replacing the cartesian product with the tensor, and conversely; in particular, $\pi_i v$ can be encoded as `match` v `with` $\langle x_1, x_2\rangle \to x_i$, even though, as said above, resources are consumed differently (sum versus upper bound). An interesting remark is that record/object calculi, where generally width subtyping is allowed, meaning that components can be discarded at runtime, and object construction happens by sequential evaluation of the fields, need to be modeled by multiplicative product and affine grades. In future work we plan to investigate object calculi which are *linear*, or, more generally, use resources in an exact way.

Terminating recursion. As anticipated, even though the type system can only derive, for recursive functions, recursion grades which are "infinite", their calls which terminate in standard semantics can terminate also in resource-aware semantics, provided that the initial amount of the function resource is enough to cover the (finite number of) recursive calls, as shown in Fig. 15.

[13]Typing rules follow the same pattern.

$$\dfrac{\dfrac{}{\mathsf{ev}\,|(1,1,z) \Rightarrow_1 \mathsf{even}|(0,1,z)}\,(\textsc{var}) \quad \dfrac{}{\mathsf{n}|(0,1,z) \Rightarrow \mathsf{z}|(0,0,z)}\,(\textsc{var}) \quad \dfrac{}{\mathtt{if\text{-}z(z,t,sn{\to}not(ev\,n))}|(0,0,z) \Rightarrow \mathsf{t}|(0,0,z)}\,(\textsc{match-l})}{\mathsf{ev\,n}|(1,1,z) \Rightarrow \mathsf{t}|(0,0,z)}\,(\textsc{app})$$

$$\dfrac{\dfrac{}{\mathsf{ev}\,|(n,1,\mathsf{s}^n z) \Rightarrow_n \mathsf{even}|(n-1,1,z)}\,(\textsc{var}) \quad \dfrac{}{\mathsf{n}|(n-1,1,z) \Rightarrow \mathsf{z}|(n-1,0,z)}\,(\textsc{var}) \quad \dfrac{}{\mathtt{if\text{-}z(z,t,sn{\to}not(ev\,n))}|(n-1,1,z) \Rightarrow b|(0,0,z)}\,(\textsc{match-l})}{\mathsf{ev\,n}|(n,1,\mathsf{s}^n z) \Rightarrow b|(0,0,z)}\,(\textsc{app})$$

Fig. 15. Example of resource-aware evaluation: terminating recursion

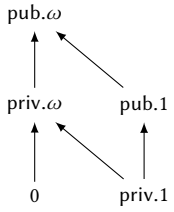

Fig. 16. Grade algebra of privacy levels and linearity

```
Result = success + failure
OptChar = none + some:Char^priv.ω
fnType= Char^priv.ω → (ok:Char^priv.ω + error)

open: String^pub.ω → FileHandle
read: FileHandle → (OptChar^priv.ω⊗ FileHandle)
write: (Char^priv.ω⊗ FileHandle) → FileHandle
close: FileHandle → Unit^0
```

Fig. 17. Types of data and filesystem interface

For brevity, we write z, s, and t for zero, succ, and true, respectively, $\mathtt{if\text{-}z}(v, e_1, \mathsf{s}x{\to}e_2)$ for $\mathtt{match}\ v\ \mathtt{with}\ \mathsf{z} \to e_1\ \mathtt{or}\ \mathsf{s}x \to e_2$, and $(r,s,\mathbf{v})$ for the environment $\mathsf{ev} : (r, \mathsf{even}), \mathsf{n} : (s, \mathbf{v})$. In the top part of the figure we show the proof tree for the evaluation of ev n in the base case, that is, in an environment where the value of n is zero. In this case, both ev and n can be graded 1, since they are used only once. In the bottom part, we show the proof tree in an environment where the value of n is $\mathsf{s}^n\mathsf{z}$, for $n \neq 0$. Here, b stands for either true or false, and $\overline{b}$ for its complement.

Processing data from/to files. The following example illustrates how we can simultaneously track privacy levels, as introduced in Example 3.2, and linearity information. Linearity grades guarantee the correct use of files, whereas privacy levels are used to ensure that data are handled without leaking information. We combine the two grade algebras with the smash product of Example 2.8. So there are five grades: 0 (meaning unused), priv.1 and pub.1 (meaning used linearly in either priv or pub mode), and priv.ω, pub.ω (meaning used an arbitrary number of times in either priv or pub mode). The partial order is graphically shown in Fig. 16. The neutral element for multiplication is pub.1, which therefore will be omitted.

In Fig. 17 are the types of the data, the processing function and the functions of a filesystem interface, assuming types Char, String, and FileHandle to be given. The type Result indicates success or failure. The type OptChar represents the presence or absence of a Char and is used in the function reading from a file; fnType is the type of a function processing a private Char and returning either a private Char or error. The signatures of the functions of the filesystem interface specify that file handlers are used in a linear way. Hence, after opening a file and doing a number of read or write operations, the file must be closed.

In the code of Fig. 18 we use, rather than $\mathtt{match}\ e_1\ \mathtt{with\ unit} \to e_2$, the alternative syntax $e_1\,;e_2$ mentioned in Section 3. The function fileRW takes as parameters an input and an output file handler and a function that processes the character read from the input file. The result indicates whether all the characters of the input file have been successfully processed and written in the output file or there was an error in processing some character.

```
1  fileRW:FileHandle →pub.ω FileHandle → fnTypepub.ω → Resultpub.ω=
2  rec fileRdWr.
3     \inF.\outF.\fn.
4        match (read inF) with <oC,inF1> ->
5           match oC with none ->
6              close inF1;close outF;success
7           or (some c) ->
8              match (fn c) with (ok c1) ->
9                 let outF1 = write <c1,outF> in
10                   fileRdWr inF1 outF1 fn
11             or error ->
12                close inF1;close outF;failure
```

Fig. 18. Processing data from/to files

The function starts by reading a character from the input file. If `read` returns `none`, then all the characters from the input file have been read and so both files are closed and the function returns `success` (lines 5—6). Closing the files is necessary in order to typecheck this branch of the match. If `read` returns a character (lines 7—12), then the processing function `fn` is applied to that character. Then, if `fn` returns a character, then this result is written to the output file using the file handler passed as a parameter and, finally, the function is recursively called with the file handlers returned by the `read` and `write` functions as arguments. If, instead, `fn` returns `error`, then both files are closed and the function returns `failure` (lines 11—12).

Observe that, given the order of Fig. 16, we have $0 \not\leq$ pub.1. Therefore the variables of type `FileHandle`, which have grade pub.1, must be used exactly once in every branch of the matches in their scope.

A call to `fileRW` could be: `fileRW (open "inFile") (open "outFile") (rec f.\x.x)`.
Note that , with

```
write: (Charpub.ω⊗ FileHandle) → FileHandle
```

the function `fileRW` would not be well-typed, since a priv character cannot be the input of `write`. On the other hand, using subsumption, we can apply `fileRW` to a processing function with type

```
Charpriv.ω → (ok:Charpub.ω + error)
```

Finally, in the type of `fileRW`, the grade of the first arrow says that the function is recursive and it is internally used in an unrestricted way. The function could also be typed with:

```
FileHandle →priv.ω FileHandle → fnTypepub.ω → Resultpriv.ω
```

However, in this case its final result would be private and therefore less usable.

7 CONCLUSION

Related work. As anticipated, the two contributions closest to this work, since they present an instrumented semantics, are [Bianchini et al. 2023b; Choudhury et al. 2021]. In [Choudhury et al. 2021], the authors develop GRAD, a graded dependent type system that includes functions, tensor products, additive sums, and a unit type. The instrumented semantics is defined on typed terms, with the only aim to show the role of the type system, whereas in [Bianchini et al. 2023b], where the underlying language is Featherweight Java, and in this paper, the definition 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. Differently from [Bianchini et al. 2023b; Choudhury et al. 2021], here we give the semantics in *big-step* style, making

it no longer necessary to annotate subterms. Moreover, we analyze in deep resource consumption in recursive functions, getting that they need to be typed with an "infinite" grade.

The type system in this paper follows the same design of those in [Bianchini et al. 2022a, 2023a,b], which, however, consider a Java-like underlying calculus. Such works also deal with two interesting issues not considered here. The former is the definition of a canonical construction leading to a unique grade algebra of *heterogeneous* grades from a family of grade algebras, thus allowing different notions of resource usage to coexist in the same program. The latter is providing linguistic support to specify *user-defined* grades, for instance grade annotations could be written themselves in Java, analogously to what happens with exceptions.

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

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

McBride [2016]; Wood and Atkey [2022] observed that contexts in a structural coeffect system form a module over the semiring of grades, event though they restrict themselves to free modules, that is, to structural coeffect systems. Recently, [Bianchini et al. 2022b] shows a significant non-structural instance, namely, a coeffect system to track sharing in the imperative paradigm.

Summary and future work. We defined, on top of a lambda calculus equipped with common constructs, a resource-aware semantics and type system, parametric on an arbitrary grade algebra. We proved resource-aware soundness, that is, that for well-typed expressions there is a computation which is not stuck for any reason, including resource consumption. The proof provides a significant, complex application of a schema previously introduced in [Ancona et al. 2017b], in a case where the semantics is non-deterministic, hence *soundness-may* needs to be proved.

As discussed in Section 1, most works on graded type systems introduce box/unbox operators in the syntax. Hence, programs typed with the type system proposed in this paper could not even be *written* in such systems. A formal comparison with a calculus/type system based on boxing/unboxing is a challenging topic for future work. However, it is not obvious how to make such a comparison, since works which present calculi with a similar expressive power to ours, e.g., [Brunel et al. 2014; Dal Lago and Gavazzo 2022], do not include an instrumented semantics, so we should as first step develop such a semantics.

An interesting fact emerged from our work is that *non-affine* grade algebras are a distinguished class; indeed, the fact that they require *exact* resource consumption poses a strong constraint, making impossible to type some constructs. It would be nice to characterize the non-affine case at the semantic level as well. That is, to state and prove a result asserting that, when the grade algebra

is non-affine, a well-typed program has a *non-wasting* reduction, in the sense that the initially available resources are all exhausted at the end.

Another direction we would like to develop is an extension of the approach to the *imperative* case, so to characterize as grades some properties which are of paramount importance in this context, such as *mutability/immutability* or *uniqueness* opposed to *linearity*, revisiting what discussed in [Marshall et al. 2022]. Note that, in the imperative case, a different notion of resource usage could be considered, e.g., rather than any time a variable occurrence needs to be replaced, any time memory needs to be accessed, even possibily distinguishing read from write accesses.

Finally, some more general topics to be investigated are type inference, for which the challenging feature are recursive functions, and combination with *effects*, as in [Dal Lago and Gavazzo 2022].

Concerning implementation, and mechanization of proofs, which of course would be beneficial developments as well, we just mention the Agda library at https://github.com/LcicC/inference-systems-agda, described in [Ciccone et al. 2021], which allows one to specify (generalized) inference systems. One of the examples provided in [Ciccone et al. 2021] is, as in this paper, a big-step semantics including divergence.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees who provided useful and detailed comments on a previous version of the paper. This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X) and has the financial support of the University of Eastern Piedmont.

A PROOFS

Lemma A.1 (Inversion for value expressions).

(1) *If* $\Gamma \vdash x : \tau^r$ *then* $x :_{r'} \tau \leq \Gamma$ *and* $r \leq r'$.
(2) *If* $\Gamma \vdash \mathtt{rec}\, f.\lambda x.e : \tau^r$ *then* $r' \cdot \Gamma' \leq \Gamma$ *and* $\tau = \tau_1^{r_1} \to_s \tau_2^{r_2}$ *such that* $\Gamma', f :_s \tau_1^{r_1} \to_s \tau_2^{r_2}, x :_{r_1} \tau_1 \vdash e : \tau_2^{r_2}$ *and* $r \leq r'$.
(3) *If* $\Gamma \vdash \mathtt{unit} : \tau^r$ *then* $\tau = \mathtt{Unit}$ *and* $\emptyset \leq \Gamma$.
(4) *If* $\Gamma \vdash \langle {}^{r_1} v_1, v_2 {}^{r_2} \rangle : \tau^r$ *then* $r' \cdot (\Gamma_1 + \Gamma_2) \leq \Gamma$, $\tau = \tau_1^{r_1} \otimes \tau_2^{r_2}$ *and* $r \leq r'$ *such that* $\Gamma_1 \vdash v_1 : \tau_1^{r_1}, \Gamma_2 \vdash v_2 : \tau_2{}^{r_2}$.
(5) *If* $\Gamma \vdash \mathtt{inl}^{r_1} v : \tau^r$ *then* $r' \cdot \Gamma' \leq \Gamma$, $\tau = \tau_1^{r_1} + \tau_2^{r_2}$ *and* $r \leq r'$ *such that* $\Gamma' \vdash v : \tau_1^{r_1}$.
(6) *If* $\Gamma \vdash \mathtt{inr}^{r_2} v : \tau^r$ *then* $r' \cdot \Gamma' \leq \Gamma$, $\tau = \tau_1^{r_1} + \tau_2^{r_2}$ *and* $r \leq r'$ *such that* $\Gamma' \vdash v : \tau_2^{r_2}$.

Lemma A.2 (Canonical Forms).

(1) *If* $\Gamma \vdash v : (\tau_1^{r_1} \to_s \tau_2^{r_2})^{r_3}$ *then* $v = \mathtt{rec}\, f.\lambda x.e$.
(2) *If* $\Gamma \vdash v : \mathtt{Unit}$ *then* $v = \mathtt{unit}$.
(3) *If* $\Gamma \vdash v : (\tau_1^{r_1} \otimes \tau_2^{r_2})^{r_3}$ *then* $v = \langle {}^{r_1} v_1, v_2 {}^{r_2} \rangle$.
(4) *If* $\Gamma \vdash v : (\tau_1^{r_1} + \tau_2^{r_2})^{r}$ *then* $v = \mathtt{inl}^{r_1} v_1$ *or* $v = \mathtt{inr}^{r_2} v_2$.

Lemma A.3 (Renaming). *If* $\Gamma, x :_{r_1} \tau_1 \vdash e : \tau_2^{r_2}$ *then* $\Gamma, x' :_{r_1} \tau_1 \vdash e[x'/x] : \tau_2^{r_2}$ *with* x' *fresh.*

Proof of Lemma 4.2. By induction on the typing rules. We show only some cases.

(T-SUB-V) We have $\Delta \vdash v : \tau^{s'}$, $\Gamma \vdash v : \tau^r$, $r \leq s'$ and $\Delta \leq \Gamma$. By induction hypothesis $s \cdot \Delta \vdash v : \tau^{s \cdot s'}$. By monotonicity $s \cdot r \leq s \cdot s'$ and $s \cdot \Delta \leq s \cdot \Gamma$, so, by (T-SUB) we get $s \cdot \Gamma \vdash e : \tau^{s \cdot r}$, that is, the thesis.

(T-VAR) By rule (T-VAR) we get the thesis.

(T-FUN) We have $\Gamma = r \cdot \Gamma'$, $v = \mathtt{rec}\, f.\lambda x.e'$, $\tau = \tau_1{}^{r_1} \to_{s'} \tau_2{}^{r_2}$ and $\Gamma', f :_{s'} \tau_1^{r_1} \to_{s'} \tau_2^{r_2}, x :_{r_1} \tau_1 \vdash e' : \tau_2^{r_2}$. By rule (T-FUN), $(s \cdot r) \cdot \Gamma' \vdash v : \tau^{s \cdot r}$. From $(s \cdot r) \cdot \Gamma' = s \cdot (r \cdot \Gamma') = s \cdot \Gamma$ we get the thesis.

(T-PAIR), (T-INL) and (T-INR) Similar to (T-FUN).

(T-SUB) We have $\Delta \vdash e : \tau^{s'}$, $\Gamma \vdash e : \tau^r$, $r \leq s'$ and $\Delta \leq \Gamma$. By induction hypothesis $s \cdot \Delta \vdash e : \tau^{s \cdot s'}$. By monotonicity $s \cdot r \leq s \cdot s'$ and $s \cdot \Delta \leq s \cdot \Gamma$, so, by (T-SUB) we get $s \cdot \Gamma \vdash e : \tau^{s \cdot r}$, that is, the thesis.

(T-APP) We have $\Gamma_1+\Gamma_2 \vdash v_1 v_2 : \tau_2^{\,r'\cdot r_2}$. By induction hypothesis $s\cdot\Gamma_1 \vdash v_1 : (\tau_1^{\,r_1} \to_{s'} \tau_2^{\,r_2})^{s\cdot(r'+r'\cdot s')}$ and $s\cdot\Gamma_2 \vdash v_2 : \tau^{s\cdot r'\cdot r_1}$. Since $s \neq \mathbf{0}$ and $r' \neq \mathbf{0}$ and, since the algebra is integral, $s\cdot r \neq \mathbf{0}$. By rule (T-APP), $s\cdot(\Gamma_1+\Gamma_2) \vdash v_1 v_2 : \tau_2^{\,s\cdot r'\cdot r_2}$, that is, the thesis.

(T-MATCH-P) By induction hypothesis $s\cdot\Gamma_1 \vdash v : (\tau_1^{\,r_1} \otimes \tau_2^{\,r_2})^{s\cdot s'}$ and $s\cdot\Gamma_2, x :_{(s\cdot s')\cdot r_1} \tau, y :_{(s\cdot s')\cdot r_2} \tau_2 \vdash e : \tau^{s\cdot r}$. Since $s \neq \mathbf{0}$ and $s' \neq \mathbf{0}$ and, since the algebra is integral, $s\cdot s' \neq \mathbf{0}$. By rule (T-MATCH-P), $s\cdot(\Gamma_1+\Gamma_2) \vdash \texttt{match}\ v\ \texttt{with}\ \langle x, y\rangle \to e : \tau^{s\cdot r}$, that is, the thesis.

□

Proof of Lemma 4.3. By case analysis on v. We show only some cases.

$v = x$ By Lemma A.1(1) $x :_{r'} \tau \leq \Gamma$ and $s\cdot r \leq r'$. Let $\Gamma' = x :_r \tau$. By monotonicity of $\cdot$ and, by transitivity of $\leq$, $(s\cdot r)\cdot x :_1 \tau = s\cdot\Gamma' \leq \Gamma$. By (T-VAR) $\Gamma' \vdash x : \tau^r$.

$v = \langle^{r_1} v_1, v_2{}^{r_2}\rangle$ By Lemma A.1(4) $r'\cdot(\Gamma_1+\Gamma_2) \leq \Gamma$ and $s\cdot r \leq r'$ and $\Gamma_1 \vdash v_1 : \tau_1^{\,r_1}$ and $\Gamma_2 \vdash v_2 : \tau_2^{\,r_2}$. Let $\Gamma' = r\cdot(\Gamma_1+\Gamma_2)$. By monotonicity of $\cdot$ and by transitivity of $\leq$ we have $s\cdot\Gamma' \leq \Gamma$. By (T-PAIR) $\Gamma' \vdash v : \tau^r$.

□

We define $\texttt{grade}(x,\Gamma) = r$ if $\Gamma = \Gamma', x :_r \tau$, otherwise $\texttt{grade}(x,\Gamma) = \mathbf{0}$.

Proof of Lemma 5.2. By induction on the syntax of v.

$v = x$ By Lemma A.1(1) $x :_{r'} \tau \leq \Delta$ with $r \leq r'$. Since $\hat\Gamma + x :_{r'} \tau + \Theta \leq \hat\Gamma + \Delta + \Theta \leq \Gamma$ and, by (T-ENV), $\rho = x_1 : (r_1, \mathbf{v}_1), \ldots, x_n : (r_n, \mathbf{v}_n)$, $\Gamma = x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n = \Gamma', x :_s \tau, \Gamma_i \vdash \mathbf{v}_i : \tau_i^{\,r_i}\ \forall i \in 1..n$ and $t_1 + r' + t_2 \leq s$ where $\texttt{grade}(x,\hat\Gamma) = t_1$ and $\texttt{grade}(x,\Theta) = t_2$. Let j be such that $x = x_j$, we get $\hat\Gamma = \hat\Gamma' + s\cdot\Gamma_j$ and $\Gamma_j \vdash \mathbf{v} : \tau^1$. By Lemma 4.2 $r\cdot\Gamma_j \vdash \mathbf{v} : \tau^r$. By rule (VAR), $x|\rho', x : (s, \mathbf{v}) \Rightarrow_r \mathbf{v}|\rho', x : (t_1+t_2, \mathbf{v})$. We have $\hat\Gamma' + (t_1+t_2)\cdot\Gamma_j + r\cdot\Gamma_j + \Theta \leq \hat\Gamma' + (t_1+t_2)\cdot\Gamma_j + r'\cdot\Gamma_j + \Theta \leq \hat\Gamma + \Theta$. We have $\texttt{grade}(y, \hat\Gamma+\Theta) \leq \texttt{grade}(y,\Gamma')$ for all $y \in \mathrm{dom}(\hat\Gamma+\Theta)\setminus\{x\}$ and $\texttt{grade}(x,\hat\Gamma+\Theta) = t_1+t_2$, so $\hat\Gamma + \Theta \leq \Gamma', x :_{t_1+t_2} \tau$. By this relation and rule (T-ENV), $\Gamma', x :_{t_1+t_2} \tau \vdash \rho' \triangleright \hat\Gamma' + (t_1+t_2)\cdot\Gamma_j$.

$v = \langle^{r_1} v_1, v_2{}^{r_2}\rangle$ By Lemma A.1(4) $r'\cdot(\Delta_1+\Delta_2) \leq \Delta$, $\tau = \tau_1^{\,r_1} \otimes \tau_2^{\,r_2}$ and $r \leq r'$ such that $\Delta_1 \vdash v_1 : \tau_1^{\,r_1}$, $\Delta_2 \vdash v_2 : \tau_2{}^{r_2}$. By Lemma 4.2 $r\cdot\Delta_1 \vdash v_1 : \tau_1{}^{r\cdot r_1}$ and $r\cdot\Delta_2 \vdash v_2 : \tau_2{}^{r\cdot r_2}$. We have $\Gamma \vdash \rho \triangleright \hat\Gamma$ with $\hat\Gamma + r\cdot\Delta_1 + (r\cdot\Delta_2 + \Theta) \leq \hat\Gamma + \Delta + \Theta \leq \Gamma$. By induction hypothesis $v_1|\rho \Rightarrow_{r\cdot r_1} \mathbf{v}_1|\rho'$, $\Delta_1' \vdash \mathbf{v}_1 : \tau^{r\cdot r_1}$ and $\Gamma_1' \vdash \rho_1' \triangleright \hat\Gamma_1'$ with $\hat\Gamma_1' + \Delta_1' + (r\cdot\Delta_2 + \Theta) \leq \Gamma_1'$. Since $\Gamma_1' \vdash \rho_1' \triangleright \hat\Gamma_1'$ with $\hat\Gamma_1' + r\cdot\Delta_2 + (\Delta_1' + \Theta) \leq \Gamma_1'$ by induction hypothesis $v_2|\rho_1' \Rightarrow_{r\cdot r_2} \mathbf{v}_2|\rho_2'$, $\Delta_2' \vdash \mathbf{v}_2 : \tau^{r\cdot r_2}$ and $\Gamma_2' \vdash \rho_2' \triangleright \hat\Gamma_2'$ with $\hat\Gamma_2' + \Delta_2' + (\Delta_1' + \Theta) \leq \Gamma_2'$. By Lemma 4.3 $\Delta_1'' \vdash \mathbf{v}_1 : \tau_1^{\,r_1}$ and $\Delta_2'' \vdash \mathbf{v}_2 : \tau_2^{\,r_2}$ with $r\cdot\Delta_1'' \leq \Delta_1'$ and $r\cdot\Delta_2'' \leq \Delta_2'$. We have $\hat\Gamma_2' + r\cdot(\Delta_1'' + \Delta_2'') + \Theta = \hat\Gamma_2' + r\cdot\Delta_1'' + r\cdot\Delta_2'' + \Theta \leq \hat\Gamma_2' + \Delta_2' + \Delta_1' + \Theta \leq \Gamma_2'$. By rules (T-PAIR) and (T-SUB-V), $r\cdot(\Delta_1'' + \Delta_2'') \vdash \langle^{r_1}\mathbf{v}_1, \mathbf{v}_2{}^{r_2}\rangle : (\tau_1^{\,r_1} \otimes \tau_2^{\,r_2})^r$. By rule (PAIR), $v|\rho \Rightarrow_r \langle^{r_1}\mathbf{v}_1, \mathbf{v}_2{}^{r_2}\rangle|\rho_2'$.

□

Lemma A.4 (Inversion for possibly diverging expressions).

(1) *If* $\Gamma \vdash \texttt{return}\ v : \tau^r$ *then* $\Gamma' \vdash v : \tau^{r'}$ *with* $r \leq r'$, $\Gamma' \leq \Gamma$ *and* $r' \neq \mathbf{0}$.

(2) *If* $\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau^r$ *then* $\Gamma_1 \vdash e_1 : \tau_1^{\,r_1}$ *and* $\Gamma_2, x :_{r_1} \tau_1 \vdash e_2 : \tau^{r'}$ *and* $\Gamma_1 + \Gamma_2 \leq \Gamma$ *and* $r \leq r'$.

(3) *If* $\Gamma \vdash v_1 v_2 : \tau_2^{\,r}$ *then* $\Gamma_1 + \Gamma_2 \leq \Gamma$ *and* $r \leq r'\cdot r_2$ *such that* $\Gamma_1 \vdash v_1 : (\tau_1^{\,r_1} \to_s \tau_2^{\,r_2})^{r'+r'\cdot s}$ *and* $\Gamma_2 \vdash v_2 : \tau_1^{\,r'\cdot r_1}$ *and* $r' \neq \mathbf{0}$.

(4) *If* $\Gamma \vdash \texttt{match}\ v\ \texttt{with}\ \texttt{unit} \to e : \tau^r$ *then* $\Gamma_1+\Gamma_2 \leq \Gamma$ *and* $r \leq t$ *such that* $\Gamma_1 \vdash v : \texttt{Unit}^{r'}$ *and* $\Gamma_2 \vdash e : \tau^t$.

(5) *If* $\Gamma \vdash \texttt{match}\ v\ \texttt{with}\ \langle x, y\rangle \to e : \tau^r$ *then* $\Gamma_1 + \Gamma_2 \leq \Gamma$ *and* $r \leq t$ *such that* $\Gamma_1 \vdash v : (\tau_1^{\,r_1} \otimes \tau_2{}^{r_2})^s$ *and* $\Gamma_2, x :_{s\cdot r_1} \tau, y :_{s\cdot r_2} \tau_2 \vdash e : \tau^t$ *and* $s \neq \mathbf{0}$.

(6) *If* $\Gamma \vdash \texttt{match}\ v\ \texttt{with}\ \texttt{inl}\ x \to e_1\ \texttt{or}\ \texttt{inr}\ x \to e_2 : \tau^r$ *then* $\Gamma_1 + \Gamma_2 \leq \Gamma$ *and* $r \leq s$ *such that* $\Gamma_1 \vdash v : (\tau_1{}^{r_1} + \tau_2{}^{r_2})^{r'}$, $\Gamma_2, x :_{r'\cdot r_1} \tau_1 \vdash e_1 : \tau^s$, $\Gamma_2, x :_{r'\cdot r_2} \tau_2 \vdash e_2 : \tau^s$ *and* $r' \neq \mathbf{0}$.

Proof of Lemma 5.11. By induction on the reduction $e|\tilde{\rho}_1 \Rightarrow \mathbf{v}|\tilde{\rho}_2$.

(RET) By Lemma A.4(1) $\Delta' \vdash v : \tau^{r'}$ with $r \preceq r'$, $\Delta' \preceq \Delta$ and $r' \neq \mathbf{0}$. By (T-SUB-V) $\Delta \vdash v : \tau^{r}$. By Lemma 5.2 $v|\rho_1 \Rightarrow_r \mathbf{v}|\rho_2'$ and $\Delta' \vdash \mathbf{v} : \tau^{r'}$ and $\Gamma' \vdash \rho_2' \triangleright \hat{\Gamma}'$ with $\hat{\Gamma}' + \Delta' \preceq_\Theta \Gamma'$. By rule (RET) $\mathtt{return}\ v|\rho_1 \Rightarrow_r \mathbf{v}|\rho_2'$. By $\mathtt{return}\ v|\rho_1 \Rightarrow_r \mathbf{v}|\rho_2'$, we derive $\mathtt{return}\ v|\tilde{\rho}_1 \Rightarrow \mathbf{v}|\tilde{\rho}_2'$. Since $\Rightarrow$ is deterministic, $\tilde{\rho}_2' = \tilde{\rho}_2$, that is, the thesis.

(LET) By Lemma A.4(2) $\Delta_1 \vdash e_1 : \tau_1^{r_1}$ and $\Delta_2, x :_{r_1} \tau_1 \vdash e_2 : \tau^{r'}$ and $\Delta_1 + \Delta_2 \preceq \Delta$ and $r \preceq r'$. We have $e_1|\tilde{\rho}_1 \Rightarrow \mathbf{v}|\tilde{\rho}_2$. We have $\Gamma \vdash \rho_1 \triangleright \hat{\Gamma}$ with $\hat{\Gamma} + (\Delta_1 + \Delta_2) \preceq_\Theta \hat{\Gamma} + \Delta \preceq_\Theta \Gamma$. By this consideration, $\hat{\Gamma} + \Delta_1 \preceq_{\Theta+\Delta_2} \Gamma$. By induction hypothesis $\Delta' \vdash \mathbf{v} : \tau_1^{r_1}$ and $\Gamma' \vdash \rho_2 \triangleright \hat{\Gamma}'$ with $\hat{\Gamma}' + \Delta' \preceq_{\Theta+\Delta_2} \Gamma'$, for some ρ_2 such that $\mathrm{erase}(\rho_2) = \tilde{\rho}_2$. By this relation we derive $\hat{\Gamma}' + \Delta_2 \preceq_{\Theta+\Delta'} \Gamma'$. By Lemma 4.3 $r_1 \cdot \Delta'' \preceq \Delta'$ and $\Delta'' \vdash \mathbf{v} : \tau_1^{\mathbf{1}}$. Since $x \notin \mathrm{dom}(\hat{\Gamma}' + \Delta_2)$ and $x \notin \mathrm{dom}(\Gamma')$ we have $\hat{\Gamma}' + \Delta_2 + x' :_{r_1} \tau_1 \preceq_{\Theta+\Delta'} \Gamma', x' :_{r_1} \tau_1$ and so $\Gamma', x' :_{r_1} \tau_1 \vdash \rho_2, x' : (r_1, \mathbf{v}) \triangleright \hat{\Gamma}' + r_1 \cdot \Delta''$. By manipulating the previous relation we have $\hat{\Gamma}' + r_1 \cdot \Delta'' + \Delta_2 + x' :_{r_1} \tau_1 \preceq_\Theta \Gamma', x' :_{r_1} \tau_1$. Since $\Delta_2, x :_{r_1} \tau_1 \vdash e_2 : \tau^{r'}$ we have $\Delta_2, x' :_{r_1} \tau_1 \vdash e_2[x'/x] : \tau^{r'}$. By induction hypothesis on $e_2[x'/x]|\tilde{\rho}_2, x' : \mathbf{v} \Rightarrow \mathbf{v}'|\rho_3$ we have $\hat{\Delta} \vdash \mathbf{v}' : \tau^{r'}$ and $\Gamma'' \vdash \rho_3 \triangleright \hat{\Gamma}''$ with $\hat{\Gamma}'' + \hat{\Delta} \preceq_\Theta \Gamma''$, for some ρ_3 such that $\mathrm{erase}(\rho_3) = \tilde{\rho}_3$, that is, the thesis.

(APP) We have $v_1|\tilde{\rho}_1 \Rightarrow \mathtt{rec}\, f.\lambda x.e|\tilde{\rho}_2$, $v_2|\tilde{\rho}_2 \Rightarrow \mathbf{v}_2|\tilde{\rho}_3$ and $e[x'/x][f'/f]|\tilde{\rho}_3, x' : \mathbf{v}_2, f' : \mathtt{rec}\, f.\lambda x.e \Rightarrow \mathbf{v}|\tilde{\rho}_4$ with x', f' fresh. By Lemma A.4(3) $\Delta_1 + \Delta_2 \preceq \Delta$ and $r \preceq r' \cdot r_2$ such that $\Delta_1 \vdash v_1 : (\tau_1^{r_1} \to_s \tau_2^{r_2})^{r' + r' \cdot s}$ and $\Delta_2 \vdash v_2 : \tau_1^{r' \cdot r_1}$ and $r' \neq \mathbf{0}$. We have $\hat{\Gamma} + \Delta_1 \preceq_{\Theta+\Delta_2} \Gamma$. By Lemma 5.2 and by Lemma A.2(1) $v_1|\rho_1 \Rightarrow_{r'+r' \cdot s} \mathtt{rec}\, f.\lambda x.e|\rho_2$ and $\Delta_1' \vdash \mathtt{rec}\, f.\lambda x.e : (\tau_1^{r_1} \to_s \tau_2^{r_2})^{r' + r' \cdot s}$ and $\Gamma' \vdash \rho_2 \triangleright \hat{\Gamma}'$ with $\hat{\Gamma}' + \Delta_1' \preceq_{\Theta+\Delta_2} \Gamma'$. Since $v|\rho \Rightarrow_t \mathbf{v}'|\rho'$ implies $v|\tilde{\rho} \Rightarrow \mathbf{v}'|\tilde{\rho}'$ and by determinism of $\Rightarrow$ and $\Rightarrow_r$ we have $\mathrm{erase}(\rho_2) = \tilde{\rho}_2$. We also have $\hat{\Gamma}' + \Delta_2 \preceq_{\Theta+\Delta_1'} \Gamma'$, so, by Lemma 5.2 $v_2|\rho_2 \Rightarrow_{r' \cdot r_1} \mathbf{v}_2|\rho_3$ and $\Delta_2' \vdash \mathbf{v}_2 : \tau_1^{r' \cdot r_1}$ and $\Gamma'' \vdash \rho_3 \triangleright \hat{\Gamma}''$ with $\hat{\Gamma}'' + \Delta_2' \preceq_{\Theta+\Delta_1'} \Gamma''$. Since $v|\rho \Rightarrow_t \mathbf{v}'|\rho'$ implies $v|\tilde{\rho} \Rightarrow \mathbf{v}'|\tilde{\rho}'$ and by determinism of $\Rightarrow$ and $\Rightarrow_r$ we have $\mathrm{erase}(\rho_3) = \tilde{\rho}_3$. By Lemma A.1(2) $r'' \cdot \Delta_1'' \preceq \Delta_1'$ such that $\Delta_1'', f :_s \tau_1^{r_1} \to_s \tau_2^{r_2}, x :_{r_1} \tau_1 \vdash e : \tau_2^{r_2}$ and $r' + r' \cdot s \preceq r''$. By these considerations we have $(r' + r' \cdot s) \cdot \Delta_1'' \preceq r'' \cdot \Delta_1'' \preceq \Delta_1'$. By rule (T-FUN) $\Delta_1'' \vdash \mathtt{rec}\, f.\lambda x.e : \tau_1^{\mathbf{1}}$. By Lemma A.3 and by Lemma 4.2 $r' \cdot (\Delta_1'', f' :_s \tau_1^{r_1} \to_s \tau_2^{r_2}, x' :_{r_1} \tau_1) \vdash e[f'/f][x'/x] : \tau_2^{r' \cdot r_2}$. By Lemma 4.3 $\Phi_2 \vdash \mathbf{v}_2 : \tau_2^{\mathbf{1}}$ with $(r' \cdot r_1) \cdot \Phi_2 \preceq \Delta_2'$. We have $\hat{\Gamma}'' + r' \cdot s \cdot \Delta_1'' + r' \cdot r_1 \cdot \Phi_2 + r' \cdot (\Delta_1'', f' :_s \tau_1^{r_1} \to_s \tau_2^{r_2}, x' :_{r_1} \tau_1) \preceq_\Theta \hat{\Gamma}'' + \Delta_2' + \Delta_1' + (f' :_{r' \cdot s} \tau_1^{r_1} \to_s \tau_2^{r_2}, x' :_{r' \cdot r_1} \tau_1)$. Since we have $\hat{\Gamma}'' + \Delta_2' \preceq_{\Theta+\Delta_1'} \Gamma''$ and $x', f' \notin \mathrm{dom}(\Gamma'' + \Delta_1' + \Delta_2')$ and $x', f' \notin \mathrm{dom}(\Gamma'')$ we have $\hat{\Gamma}'' + \Delta_2' + \Delta_1' + (f' :_{r' \cdot s} \tau_1^{r_1} \to_s \tau_2^{r_2}, x' :_{r' \cdot r_1} \tau_1) \preceq_\Theta \Gamma'', x' :_{r' \cdot r_1} \tau_1, f' :_{r' \cdot s} (\tau_1^{r_1} \to_s \tau_2^{r_2})$. By rule (T-ENV) $\Gamma'', x' :_{r' \cdot r_1} \tau_1, f' :_{r' \cdot s} (\tau_1^{r_1} \to_s \tau_2^{r_2}) \vdash \rho_3, x' : (r' \cdot r_1, \mathbf{v}_2), f' : (r' \cdot s, \mathtt{rec}\, f.\lambda x.e) \triangleright \hat{\Gamma}'' + r' \cdot s \cdot \Phi_1 + r' \cdot r_1 \cdot \Phi_2 + r' \cdot (\Delta_1'', f' :_s \tau_1^{r_1} \to_s \tau_2^{r_2}, x' :_{r_1} \tau_1)$. By induction hypothesis on $e[x'/x][f'/f]|\tilde{\rho}_3, x' : \mathbf{v}_2, f' : \mathtt{rec}\, f.\lambda x.e \Rightarrow \mathbf{v}|\tilde{\rho}_4$ we get the thesis.

(MATCH-P) By Lemma A.4(5) $\Delta_1 + \Delta_2 \preceq \Delta$ and $r \preceq t$ such that $\Delta_1 \vdash v : (\tau_1^{r_1} \otimes \tau_2^{r_2})^s$ and $\Delta_2, x :_{s \cdot r_1} \tau, y :_{s \cdot r_2} \tau_2 \vdash e : \tau^t$ and $s \neq \mathbf{0}$. By Lemma 5.2 $v|\rho \Rightarrow_s \mathbf{v}|\rho'$ and $\Delta_1' \vdash \mathbf{v} : \tau^r$ and $\Gamma' \vdash \rho_1 \triangleright \hat{\Gamma}'$ with $\hat{\Gamma}' + \Delta_1' + \Delta_2 \preceq \Gamma'$. By Lemma A.2(3) $\mathbf{v} = \langle^{r_1} \mathbf{v}_1, \mathbf{v}_2\, ^{r_2}\rangle$. By Lemma A.1(4) $r' \cdot (\hat{\Delta}_1 + \hat{\Delta}_2) \preceq \Delta_1'$ and $s \preceq r'$ such that $\hat{\Delta}_1 \vdash \mathbf{v}_1 : \tau_1^{r_1}$ and $\hat{\Delta}_2 \vdash \mathbf{v}_2 : \tau_2^{r_2}$. By Lemma 4.3 $\Phi_1 \vdash \mathbf{v}_1 : \tau_1^{\mathbf{1}}$ and $\Phi_2 \vdash \mathbf{v}_2 : \tau_2^{\mathbf{1}}$ with $r_1 \cdot \Phi_1 \preceq \hat{\Delta}_1$ and $r_2 \cdot \Phi_2 \preceq \hat{\Delta}_2$. By Lemma A.3 and rule (T-SUB) $\Delta_2, x' :_{s \cdot r_1} \tau_1, y' :_{s \cdot r_2} \tau_2 \vdash e[x'/x][y'/y] : \tau^r$. We have $\hat{\Gamma}' + (s \cdot r_1) \cdot \Phi_1 + (s \cdot r_2) \cdot \Phi_2 + (\Delta_2, x :_{s \cdot r_1} \tau_1, y :_{s \cdot r_2} \tau_2) \preceq \hat{\Gamma}' + s \cdot \hat{\Delta}_1' + s \cdot \hat{\Delta}_2 + (\Delta_2, x :_{s \cdot r_1} \tau_1, y :_{s \cdot r_2} \tau_2) \preceq \hat{\Gamma}' + \Delta_1' + (\Delta_2, x :_{s \cdot r_1} \tau_1, y :_{s \cdot r_2} \tau_2)$. Since $x', y' \notin \mathrm{dom}(\hat{\Gamma}' + \Delta_1' + \Delta_2)$ and $x', y' \notin \mathrm{dom}(\Gamma')$ and $\hat{\Gamma} + \Delta_1' + \Delta_2 \preceq \Gamma$, we have $\hat{\Gamma}' + \Delta_1' + (\Delta_2, x :_{s \cdot r_1} \tau_1, y :_{s \cdot r_2} \tau_2) \preceq \Gamma', x :_{s \cdot r_1} \tau_1, y :_{s \cdot r_2} \tau_2$. By (T-ENV) $\Gamma', x :_{s \cdot r_1} \tau_1, y :_{s \cdot r_2} \tau_2 \vdash \rho_1, x' : (s \cdot r_1, \mathbf{v}_1), y' : (s \cdot r_2, \mathbf{v}_2) \triangleright \hat{\Gamma}' + (s \cdot r_1) \cdot \Phi_1 + (s \cdot r_2) \cdot \Phi_2$. By induction hypothesis on $e[x'/x][y'/y]|\rho_1, x' : (s \cdot r_1, \mathbf{v}_1), y' : (s \cdot r_2, \mathbf{v}_2) \Rightarrow \mathbf{v}'|\rho_2$ we get the thesis.

□

REFERENCES

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

Davide Ancona, Francesco Dagnino, and Elena Zucca. 2017a. Generalizing Inference Systems by Coaxioms. In *European Symposium on Programming, ESOP 2017 (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, Berlin, 29–55. https://doi.org/10.1007/978-3-662-54434-1_2

Davide Ancona, Francesco Dagnino, and Elena Zucca. 2017b. Reasoning on Divergent Computations with Coaxioms. *Proceedings of ACM on Programming Languages* 1, OOPSLA (2017), 81:1–81:26. https://doi.org/10.1145/3133905

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

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2022a. A Java-Like Calculus with User-Defined Coeffects. In *ICTCS'22 - Italian Conf. on Theoretical Computer Science*, Ugo Dal Lago and Daniele Gorla (Eds.), Vol. 3284. CEUR-WS.org, 66–78. https://ceur-ws.org/Vol-3284/8563.pdf

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2023a. A Java-like calculus with heterogeneous coeffects. *Theoretical Computer Science* 971 (2023), 114063. https://doi.org/10.1016/j.tcs.2023.114063

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2023b. Multi-Graded Featherweight Java. In *European Conference on Object-Oriented Programming, ECOOP 2023 (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 3:1–3:27. https://doi.org/10.4230/LIPIcs.ECOOP.2023.3

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

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

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

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

Luca Ciccone, Francesco Dagnino, and Elena Zucca. 2021. Flexible Coinduction in Agda. In *ITP 2021 - International Conference on Interactive Theorem Proving (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 13:1–13:19. https://doi.org/10.4230/LIPIcs.ITP.2021.13

Patrick Cousot and Radhia Cousot. 1992. Inductive Definitions, Semantics and Abstract Interpretations. In *ACM Symposium on Principles of Programming Languages, POPL 1992*, Ravi Sethi (Ed.). ACM Press, New York, 83–94. https://doi.org/10.1145/143165.143184

Francesco Dagnino. 2019. Coaxioms: flexible coinductive definitions by inference systems. *Logical Methods in Computer Science* 15, 1 (2019). https://doi.org/10.23638/LMCS-15(1:26)2019

Francesco Dagnino. 2022. A Meta-theory for Big-step Semantics. *ACM Trans. Comput. Log.* 23, 3 (2022), 20:1–20:50. https://doi.org/10.1145/3522729

Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. 2020. Soundness Conditions for Big-Step Semantics. In *European Symposium on Programming, ESOP 2020 (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 169–196. https://doi.org/10.1007/978-3-030-44914-8_7

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

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

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

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

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Information and Computation* 207, 2 (2009), 284–304. https://doi.org/10.1016/j.ic.2007.12.004

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

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

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

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

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

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

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

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Received 2023-04-14; accepted 2023-08-27