Università di Genova

Dipartimento di
Informatica, Bioingegneria,
Robotica e Ingegneria dei Sistemi

Resource-awareness for Java-like languages and beyond

Riccardo Bianchini

Theses Series DIBRIS-TH-2024-42

Ph.D. Thesis

Università di Genova
Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi
Ph.D. Thesis in
Computer Science and System Engineering
Computer Science Curriculum

Resource-awareness for Java-like languages and beyond

by

Riccardo Bianchini

January 2024

Ph.D. Thesis in Computer Science and System Engineering (S.S.D. INF/01)
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università di Genova

Candidate
Riccardo Bianchini
riccardo.bianchini@edu.unige.it

Title
Resource-awareness
for Java-like languages and beyond

Advisors
Elena Zucca
DIBRIS, Università di Genova
elena.zucca@unige.it

Francesco Dagnino
DIBRIS, Università di Genova
francesco.dagnino@dibris.unige.it
Paola Giannini
DiSSTE, Università del Piemonte Orientale
paola.giannini@uniupo.it

External Reviewers
Ugo Dal Lago
Dipartimento di Informatica - Scienza e Ingegneria, Università di Bologna
ugo.dallago@unibo.it

Harley Eades III
School of Computer and Cyber Sciences, Augusta University
harley.eades@gmail.com

Stephanie Weirich
School of Engineering and Applied Science, University of Pennsylvania
sweirich@seas.upenn.edu

Location
DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy

Submitted On
January 2024

Abstract

Reasoning about programs and their correctness concerns, in the first place, their input/output behaviour. However, there are many important properties which are non-extensional, that is, allow to further classify programs. Among those, a significant class are the properties related to the resources needed to carry out the computation successfully. There are many possible views of what a "resource" is, e.g., space or time complexity; in this thesis, a resource is meant to be some external data, used in a program through an internal name (a variable), and *resource-awareness* means the ability to track, statically and/or at runtime, how these resources are used by the program.

The aim of this thesis is to provide design guidelines and formal foundations to smoothly add resource-awareness to a programming language, mainly focusing on the object-oriented paradigm. This is a novelty, since in the literature resource-awareness has been studied in the context of functional languages. To achieve this goal, the key idea is to use annotations, called *grades*, which, intuitively, represent the availability of a resource. These annotations are elements of an algebraic structure called *grade algebra*, and both type system and reduction are given parametrically on an arbitrary grade algebra, modeling a particular kind of usages. We also investigate the possibility for to the programmer to define her/his grades. The thesis includes two chapters which achieve additional results, notably the application of the proposed approach to a functional language, a novel formulation of resource-aware semantics, and an application of grades to a challenging case, that is, to characterize sharing and immutability properties in the context of imperative languages.

Acknowledgements

I am very grateful to my supervisors, Elena Zucca, Francesco Dagnino and Paola Giannini. Your advice and guidance have been essential for the development of the results of this thesis and all through my PhD. Basically, you taught me how science works. I am also immensely grateful to you for availability, patience and also kindness towards me.

I am really thankful to the members of my thesis committee for their insightful feedback and constructive criticisms that immensely contributed to the refinement of this research.

Special thanks go to my parents, Stefania and Marco, and to my sister, Francesca. You have been an unwavering and moral support and encouragement throughout my academic journey. Your love and belief in my abilities have been a constant source of strength.

My sincere appreciation goes to all my colleagues and friends in the DoCS (Dottorandi in Computer Science). In particular, among others, I want to thank Marco, Luca, Eros, Marianna, Matteo, Federico, Lorenzo, Andrea, Pietro and, obviously, Francesco: your companionship, stimulating discussions, and shared wisdom enriched and made this journey much funnier.

Lastly, I am grateful to all the authors and researchers whose works have guided and informed my study. This thesis would not have been possible without the contributions and support of all these individuals. Thank you.

Contents

1

Introduction

Recent research has devoted an increasing interest to *resource-awareness*, that is, to formal techniques for reasoning about the usage of resources in computations. The aim is to compare programs not only by considering what is computed, but also how it is computed; that is, to consider some non-extensional properties of programs, by a quantitative analysis. There are many possible views of what a "resource" is, such as space, time, etc.; in this thesis, a resource is an external, that is, not generated by the program itself, data structure. In programs we associate names to data structures, to refer to them, so the most natural way to model resources is to track the usage of free variables. Resource-awareness can be accomplished statically, by a resource-aware type system, and/or dynamically, by a resource-aware semantics.

RESOURCE-AWARE TYPE SYSTEMS The aim of resource-aware type systems is to statically approximate not only the expected result type of a program, but also the way resources are used. As a concrete example, we would like to distinguish the functions $\lambda x.5$, $\lambda x.x$, and $\lambda x.x + x$, since they use their parameter 0, 1, and 2 times in their body, respectively, whereas a standard type system would assign the same type to all three.

The idea of resource-aware type systems originates from linear logic and linear type systems. Linear logic, introduced by Girard in his seminal paper [34], treats logical statements as resources, which cannot be duplicated or discarded. Analogously, in linear type systems, variables are used exactly once; they are a kind of *substructural* type systems, so called since some *structural* rules admissible in standard type systems are disallowed, notably weakening and contraction, as will be detailed in Section 2.2. The ability to duplicate and discard resources can be recovered by the *exponential* modality !, also called "bang". Using this mechanism, it is possible to distinguish variables used exactly once from those used without constraints. As next step, we can refine the bang modality to keep track exactly of how many times a resource is used. This is achieved by considering a family of modalities $!_n$ indexed by (extended) natural numbers and by formulating appropriate weakening and contraction rules, to correctly handle indices. The next step is to realise that natural numbers are just an instance of an algebraic structure. This leads to a further generalization obained by considering modalities indexed by *grades*, that is, elements of a given algebraic structure, basically a semiring specifying *sum* $+$, *multiplication* $\cdot$, $\mathbf{0}$ and $\mathbf{1}$ constants, and some kind of order relation. In

this way, we can track an arbitrary notion of resource usage, just by choosing a specific semiring of grades.

Type-and-coeffect systems, or simply *coeffect systems*, are a recently proposed form of type systems which is resource-aware. They are, in a sense, the dual of effect systems: given a generic type judgment $\Gamma \vdash e : \tau$, effects can be seen as an enrichment of the type τ, modeling side effects of the execution, whereas coeffects can be seen as an enrichment of the context Γ, modeling how execution needs to use external resources.

Starting from a usual typing judgement, of shape $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$, where we track only the type of variables, we add grades, called *coeffects* when used in this position, to track also usage of variables, obtaining the judgment:

$$x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \vdash e : \tau$$

The intuition is that coeffect r_i models how variable x_i is used in expression e. In the example above, we could use as coeffects natural numbers, to count how many times parameter x is used in function bodies, that is, 0, 1, 2, respectively.

Graded (modal) type systems extend type-and-coeffect systems by tracking not only how variables are used in e, but also how the result of e should be used in a program context. In this thesis, this will be expressed by a judgment:

$$x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \vdash e : \tau^r$$

where we also add a grade annotation to the expression type. Since, as it will be described in Section 2.4, grades belong to a semiring, commonly indicated by R, for grades we use the meta-variable r.

RESOURCE-AWARE SEMANTICS (AND SOUNDNESS) The basic feature of resource-aware semantics is that substitution of variables is not performed once and for all, as in the standard β-rule, but each variable occurrence is replaced when needed, by also decrementing the availability of the associated resource in some way. A precursor of this idea can be considered the lambda-calculus with molteplicities by Boudol [13], where expressions include environments associating to each variable a bag (multiset) of resources; we can replace a variable only if there is at least one resource in the bag.

Resource-aware semantics as in this thesis has been inspired by the recent work in [18], where expressions are reduced in an external environment which associates to variables, besides a value, a grade, as in resource-aware type systems. Reduction is stuck if a certain usage is not allowed.

By having resource-awareness both in type system and semantics, it is possible to express and prove a *resource-aware soundness* theorem. This theorem not only guarantees standard soundness, but also that a well-typed program has a computation which does not get stuck due to resource consumption.

RESOURCE-AWARENESS FOR JAVA-LIKE LANGUAGES AND BEYOND The main aim of the thesis is the design of a resource-aware extension for Java-like languages. To this end we provide, for a paradigmatic Java-like calculus, resource-aware semantics and graded type system, both parametric on arbitrary grades, and prove resource-aware soundness. We also show that the

language can support different kinds of grades, even user-defined. The thesis includes two chapters which go beyond such main aim, as detailed below.

OUTLINE AND RELATION WITH PUBLISHED PAPERS

CHAPTER 2 After fixing some terminology and notations used throughout the thesis, we provide a gentle introduction to coeffect systems and graded type systems. Moreover, we formally define *grade algebras*, the structures we use to model usage of resources.

CHAPTER 3 We present a simple Java-like calculus equipped with grades. Notably, we provide a resource-aware semantics and type system, and prove resource-aware soundness. The content of this chapter is taken from

> Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca. Multi-graded Featherweight Java. ECOOP 2023 [10]

In [7], not included in the thesis, it is shown that resource-aware semantics can be naturally seen as a kind of monitored reduction.

CHAPTER 4 The aim is to allow different kinds of grades to be used in the same program. To this end, we provide the construction of a grade algebra of *heterogeneous grades* from given grade algebras and homomorphisms between them. Moreover, we allow the programmer to define her/his grades, by writing *grade classes*, implementing methods corresponding to the ingredients of a grade algebra, and *homomorphism classes*, implementing the homomorphisms between grade algebras. Grade annotations are values of grade classes. The content of this chapter is partly reworked from the following papers:

> Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca. A Java-like calculus with user-defined coeffects. ICTCS 2022 [9]

> Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca. A Java-like calculus with heterogeneous coeffects. TCS 2023 [8]

> Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca. Multi-graded Featherweight Java. ECOOP 2023 [10]

More precisely, the construction of the grade algebra of heterogeneous grades is as by Bianchini et al. [10]. The derivation of grade algebras and homomorphisms from programmer's code, instead, largely extends that by Bianchini et al. [8, 9], where only coeffects are considered, combining grades of different kinds always leads to the trivial grade, and implementation is directly expressed in Java code rather than also in an extended Java-like calculus as done here.

CHAPTER 5 We go beyond the object-oriented paradigm and define a functional language equipped with grades, posing additional challenges, such as higher-order functions and (recursive) structural types. Moreover, we go beyond small-step semantics, where subterms need to be annotated to ensure that their reduction happens at each step with the same grade, describing the resource-aware semantics in *big-step* style, so that no annotations are needed. Since in big-step semantics non-terminating and stuck computations are indistinguishable, we extend the big-step judgment to model divergence explicitly by using a *generalized inference system*, where rules are interpreted in an *essentially coinductive* way. The content of this chapter is taken from

> Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca. Resource-aware soundness for big-step semantics. OOPSLA 2023 [11]

CHAPTER 6 We go beyond *structural coeffect systems*, considered in the previous chapters, where the coeffect of each single variable is computed independently, by considering a significant example in which they are not adequate. Notably, we want to use coeffects to statically guarantee relevant properties on the usage of memory in an imperative language. Indeed, the fact that a program introduces sharing between two variables, say x and y, for instance through a field assignment $x.f = y$ in an object-oriented language, clearly has a coeffect (grade) nature, being a particular way to use the resources x and y. However, this sharing information cannot be tracked per-variable. The content of this chapter is taken from

> Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, Marco Servetto. Coeffects for sharing and mutation. OOPSLA 2022 [12]

2

Resource-awareness

After fixing in Section 2.1 some terminology and notations used throughout the thesis, in Section 2.2 and Section 2.3 we provide a gentle introduction, first to *(type-and-)coeffect systems*, an enriched form of type systems which tracks the way external resources (variables) are used in an expression, and then to *graded type systems*, where we also track the way results of expressions are used in a program context. In these sections we use the notion of grade in a rather general way; in Section 2.4, we formally define *grade algebras*, the structures we use to model usage of resources.

2.1 Terminology and notations

Elements of a grade algebra are called *coeffects* when they are used in a context, on the left-hand side of a typing judgment, *grades* when they are used as annotations of types, on the right-side. We use the metavariables σ and τ for standard (that is, non-graded) types, and the metavariables S and T for graded types. Given e, e' expressions and x variable, we denote by $e[e'/x]$ the usual capture-avoiding substitution, and given two sequences representing maps, e.g., contexts Δ and Γ, with disjoint domains, we denote by Δ, Γ their concatenation (union of maps).

2.2 Coeffect systems

To illustrate how a (type-and-)coeffect system works, we start from a simple example, namely, a coeffect system for the call-by-name simply-typed λ-calculus, where we trace when a variable is either not used, or used linearly (that is, exactly once), or used in an unrestricted way, as expressed by assigning to the variable a *coeffect* r which is either 0, or 1, or ∞. In Figure 2.1 we report reduction and the standard type system, and in Figure 2.2 the coeffect system.

A *coeffect context* γ, of shape $x_1 : r_1, \ldots, x_n : r_n$, where order is immaterial and $x_i \neq x_j$ for $i \neq j$, represents a map from variables to coeffects where only a finite number of variables have non-zero coeffect. A *(type-and-coeffect) context*, of shape $\Gamma = x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n$, with analogous conventions, represents the pair of the standard type context $x_1 : \tau_1 \ldots, x_n : \tau_n$, and the coeffect context $x_1 : r_1, \ldots, x_n : r_n$.

$$
\begin{array}{lcl}
e & ::= & n \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2 \\
\tau & ::= & \mathsf{int} \mid \tau_1 \to \tau_2 \\
\Gamma, \Delta & ::= & x_1 : \tau_1, \ldots, x_n : \tau_n
\end{array}
$$

$$
(\textsc{AppAbs}) \quad \frac{}{(\lambda x{:}\tau.e)e' \to e[e'/x]} \qquad (\textsc{App}) \quad \frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2}
$$

$$
(\textsc{t-const}) \quad \frac{}{\Gamma \vdash n : \mathsf{int}} \qquad (\textsc{t-var}) \quad \frac{}{\Gamma \vdash x : \tau}\ \Gamma(x) = \tau
$$

$$
(\textsc{t-abs}) \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \qquad (\textsc{t-app}) \quad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_1}
$$

FIGURE 2.1 Simply-typed lambda calculus

$$
\begin{array}{lcl}
e & ::= & n \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2 \\
r & ::= & 0 \mid 1 \mid \infty \\
\tau & ::= & \mathsf{int} \mid \tau_1 \xrightarrow{r} \tau_2 \\
\gamma & ::= & x_1 : r_1, \ldots, x_n : r_n \\
\Gamma, \Delta & ::= & x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n
\end{array}
$$

$$
(\textsc{AppAbs}) \quad \frac{}{(\lambda x{:}\tau.e)e' \to e[e'/x]} \qquad (\textsc{App}) \quad \frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2}
$$

$$
(\textsc{t-const}) \quad \frac{}{\emptyset \vdash n : \mathsf{int}} \qquad (\textsc{t-var}) \quad \frac{}{x :_1 \tau \vdash x : \tau}
$$

$$
(\textsc{t-sub}) \quad \frac{\Gamma \vdash e : \tau}{\Gamma' \vdash e : \tau}\ \Gamma \preceq \Gamma' \qquad (\textsc{t-abs}) \quad \frac{\Gamma, x :_r \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \xrightarrow{r} \tau_2}
$$

$$
(\textsc{t-app}) \quad \frac{\Gamma_1 \vdash e_1 : \tau_2 \xrightarrow{r} \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1 + r \cdot \Gamma_2 \vdash e_1 e_2 : \tau_1}
$$

FIGURE 2.2 A simple structural coeffect system

In literature, there are many definitions of coeffects/grades [2, 5, 16, 18, 30, 31, 46, 49, 56], generally assuming that they form a semiring, that is, are equipped with a *sum* $+$, and a *multiplication* $\cdot$, satisfying some natural axioms, and moreover some form of order relation. In the thesis, we assume a variant of such definitions called a *grade algebra*, formally defined in Section 2.4, providing a partial order $\preceq$ besides sum and multiplication. In the example, the (pretty intuitive) definition of such a structure is given below.

$0 \preceq \infty, 1 \preceq \infty$

$+$	0	1	∞
0	0	1	∞
1	1	∞	∞
∞	∞	∞	∞

$\cdot$	0	1	∞
0	0	0	0
1	0	1	∞
∞	0	∞	∞

The typing rules use three operators on contexts: *partial order* $\preceq$, *sum* $+$ and *scalar multiplication* $\cdot$ of a coeffect with a context, which are defined by first taking, on coeffect contexts, the pointwise application of the corresponding operator on coeffects, and then lifting to type-and-coeffect contexts, as will be formally defined at the end of Section 2.4 and in Section 3.3. Note that when lifted to type-and-coeffect contexts the sum becomes partial, since we require a common variable to have the same type.

In rule (T-CONST) no variable is used. In rule (T-VAR), only the given variable is used, exactly once. An alternative formulation found in literature is

$$\text{(T-VAR)}\ \frac{}{0 \cdot \Gamma + x :_1 \tau \vdash x : \tau}$$

where the coeffect context is one of those representing the map where the given variable is used exactly once, and no other is used. Indeed, $0 \cdot \Gamma$ is a context where all variables have 0 coeffect. This generalization is already provided in our presentation, since we include a subsumption rule (T-SUB), allowing a well-typed expression to be typed in a less specific context, where coeffects are overapproximated. This rule is also useful, e.g., in the presence of a conditional construct, as its typing rule usually requires the two branches to be typed in the same context (that is, to use resources in the same way) and subsumption relaxes this condition. In literature this rule is often contravariant in the context, see, e.g., Gaboardi et al. [30]. This happens since the order on contexts is defined as the pointwise extension of subtyping, that is, models the strength of type assumptions: the smaller the context is, the stronger these assumption are. Here we do not consider subtyping, and order is the pointwise extension of order on grades, that is, models the amount of resources: the larger the context is, more available resources we have.

In rule (T-ABS), the type of a lambda expression is decorated with the coeffect assigned to the binder when typechecking the body. In rule (T-APP), the coeffects of an application are obtained by summing the coeffects of the first subterm, which is expected to have a functional type decorated with a coeffect, and the coeffects of the argument multiplied by the decoration of the functional type.

Let us compare the standard type system in Figure 2.1 and the coeffect system in Figure 2.2. In the former, when typechecking an expression, the

context is just propagated top-down to subterms, in rule (T-CONST) there is no requirement on the context, and in rule (VAR) the only requirement is that the variable should be present. Consequently, the following *weakening* and *contraction* rules turn out to be admissibile:

$$\text{(WEAK)}\ \frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau} \qquad \text{(CONTR)}\ \frac{\Gamma, x : \tau', y : \tau' \vdash e : \tau}{\Gamma, z : \tau' \vdash e[z/x][z/y] : \tau}$$

Rule (WEAK) states that, in typechecking an expression, we can always add useless resources, whereas rule (CONTR) states that we can unify resources with the same type. As a consequence, looking at the typing context, we have no information about the effective usage of variables, since they may have been discarded by (WEAK) or duplicated by (CONTR). In a coeffect system, instead, when typechecking an expression, the context is computed bottom-up, starting from the axioms. Thus, the above rules take the following form:

$$\text{(COEFF-WEAK)}\ \frac{\Gamma \vdash e : \tau}{\Gamma, x :_r \tau' \vdash e : \tau}\ 0 \preceq r \qquad \text{(COEFF-CONTR)}\ \frac{\Gamma, x :_r \tau', y :_s \tau' \vdash e : \tau}{\Gamma, z :_{r+s} \tau' \vdash e[z/x][z/y] : \tau}$$

We can add a new variable, but only if its grade is greater or equal than 0; for instance, we cannot safely add a linear variable, since it should necessarily be used once. We can unify two resources with the same type, but keeping track of their usage by assigning to the new variable the sum of the coeffects of the old ones. Therefore, the typing context provides complete information about the usage of all variables.

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

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

Note also that, by just changing the grade algebra, we obtain a different coeffect system. For instance, an easy variant is to consider as coeffects the natural numbers, with, as partial order, the equality, and as sum and multiplication the standard ones, tracking *exactly how many times* a variable is used. The definition of contexts and their operations, and the typing rules, can be kept exactly the same.

2.3 Graded type systems

With coeffect systems, we can track how resources (variables) are used in an expression. However, we would like to also track how the result of an

expression is used in a program context; for instance, in a Java-like language, how many times the result of a method should be used by a client. To this end, we can decorate types with grades, obtaining a *graded type system*. Our approach is novel with respect to that generally used in the literature on graded types. Notably, in such works the production of types is $\tau ::= \ldots \mid \Box_r \tau$, that is, grade decorations can be arbitrarily nested. Correspondingly, the syntax includes an explicit *box* construct, which transforms a term of type τ into a term of type $\Box_r \tau$, through a *promotion* rule which multiplies the context with r, and a corresponding unboxing mechanism, as illustrated below by rules inspired by [2].

$$\text{(PROM)}\ \frac{\Gamma \vdash e : \tau}{r \cdot \Gamma \vdash [e] : \Box_r \tau} \qquad \text{(UNBOX)}\ \frac{\Gamma \vdash e : \Box_r \tau \quad \Delta, x :_{s \cdot r} \tau \vdash e' : \tau'}{s \cdot \Gamma + \Delta \vdash \texttt{let}\ [x] = e\ \texttt{in}\ e' : \tau'}$$

The aim of the thesis is to design resource-aware semantics and type system, and formulate and prove resource-aware soundness, in an as much as possible light, abstract and general way, *without requiring ad-hoc changes* to the underlying language. Hence, we prefer a much lighter approach, likely more convenient for Java-like languages, where the syntax of terms is not affected. The production for types is $T ::= \tau^r$, that is, all types are (once) graded; in contexts, types are non-graded, and grades are used as coeffects, leading to a judgment of shape $x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \vdash e : \tau^r$. Moreover, since there is no boxing/unboxing, there is no explicit promotion rule, but different grades can be assigned to an expression, assuming different coeffect contexts. In other words, promotion is applied "locally".

The fact that syntax is unaffected 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
- ensuring backward compatibility, since old code will still work, by embedding plain types into graded types. A simple way to do this is to assume a top grade and see plain types as graded with such top. For instance, with grades $r ::= 0 \mid 1 \mid \infty$, non-graded code can be seen as ∞-graded.

A precise definition of the algebraic structure we require on grades is provided in the next section.

2.4 Grade algebras and coeffect contexts

In this section we introduce the algebraic structures we will use throughout the thesis. 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.

DEFINITION AND EXAMPLES Our definition is a slight variant of others in literature [2, 5, 16, 18, 30, 31, 46, 49, 56], which are all instances of ordered semirings.

DEFINITION 2.4.1 (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 $r, r' s, s' \in |R|$;

An ordered semiring is a semiring with a partial order on its carrier which makes sum and multiplication monotonic with respect to it. Roughly, sum 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 2.4.2 (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 2.4.3 :

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, sum 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 sum is idempotent, but rather express how/in which mode a resource can be used. They are called *informational* in [2].

6. Given grade algebras $R = \langle |R|, \leq_R, +_R, \cdot_R, \mathbf{0}_R, \mathbf{1}_R\rangle$ and $S = \langle |S|, \leq_S, +_S, \cdot_S, \mathbf{0}_S, \mathbf{1}_S\rangle$, the *product* $R \times S = \langle \{\langle r, s\rangle \mid r \in |R| \ \wedge \ s \in |S|\}, \leq, +, \cdot, \langle \mathbf{0}_R, \mathbf{0}_S\rangle, \langle \mathbf{1}_R, \mathbf{1}_S\rangle\rangle$, where operations are the pairwise application of the operations for R and S, is a grade algebra.

7. Given a grade algebra $R = \langle |R|, \leq_R, +_R, \cdot_R, \mathbf{0}_R, \mathbf{1}_R\rangle$, set $\mathsf{Ext}\, R = \langle |R| + \{\infty\}, \leq , +, \cdot, \mathbf{0}_R, \mathbf{1}_R\rangle$ where $\leq$ extends $\leq_R$ by adding $r \leq \infty$ for all $r \in |\mathsf{Ext}\, R|$ and $+$ and $\cdot$ extend $+_R$ and $\cdot_R$ by $r + \infty = \infty + r = \infty$, for all $r \in |\mathsf{Ext}\, R|$, and $r \cdot \infty = \infty \cdot r = \infty$, for all $r \in |\mathsf{Ext}\, R|$ with $r \neq \mathbf{0}_R$, and $\mathbf{0}_R \cdot \infty = \infty \cdot \mathbf{0}_R = \mathbf{0}_R$. Then, $\mathsf{Ext}\, R$ is a grade algebra, 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 $\langle r, s\rangle \leq \langle r', s'\rangle$ iff $r' \preceq_R r$ and $s \preceq_R s'$. Then, $\mathsf{Int}(R) = \langle |\mathsf{Int}(R)|, \leq, +, \cdot, \langle \mathbf{0}_R, \mathbf{0}_R\rangle, \langle \mathbf{1}_R, \mathbf{1}_R\rangle\rangle$ is a grade algebra, with $+$ and $\cdot$ defined pointwise.

As a more concrete example we can think of a file that, after being opened, should be closed. This check could be done with the grade algebra in Example 2.4.3(2) by annotating the file with coeffect 1.

TAXONOMY Grade algebras can be further classified depending whether they satisfy some properties, as detailed below.

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 $\mathbf{1} \leq \mathbf{0}$, as this implies $r \leq \mathbf{0}$, hence $r = \mathbf{0}$, for all $r \in |R|$, by the axioms of ordered semiring and Definition 2.4.2.

A grade algebra is still a quite wild structure, notably there are weird phenomena due to the interaction of sum 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 2.4.4 : Let $R = \langle |R|, \leq, +, \cdot, \mathbf{0}, \mathbf{1}\rangle$ 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 2.4.3 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 2.4.5 : 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:

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

It is easy to check that the following proposition holds.

PROPOSITION 2.4.6 : 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 2.4.7 : Let R and S be non-trivial, reduced and integral grade algebras. The *smash product* of R and S is $R \mathbin{⍉} S = \langle |R \mathbin{⍉} S|, \preceq, +, \cdot, \hat{\mathbf{0}}, \langle \mathbf{1}_R, \mathbf{1}_S\rangle\rangle$, where $|R⍉S| = |(R\times S)_{\mathbf{0}}| \setminus (|R|\times\{\mathbf{0}_S\}\cup\{\mathbf{0}_R\}\times|S|)$, $\preceq$, $+$ and $\cdot$ are the restrictions of the order and operations of $(R \times S)_{\mathbf{0}}$, as in Definition 2.4.5, to the subset $|R \mathbin{⍉} S|$, and $\hat{\mathbf{0}}$ is the zero of $(R \times S)_{\mathbf{0}}$. It is easy to see $R \mathbin{⍉} S$ is a non-trivial, reduced and integral grade algebra.

Finally, an important classification of grade algebras depends on the existence of grades which are non-comparable with $\mathbf{0}$. Indeed, through weakening, see rule (COEFF-WEAK) in Section 2.3, an expression which does not use a resource (variable) x, can be typechecked in a context where x has a grade r such that $0 \preceq r$. In this context, the grade represents a usage which is available, but *discarded* by the program. For instance, in the linearity grade algebra of Example 2.4.3(2), the element ∞ can be discarded, while the element 1 cannot. This remark leads to the following definition.

DEFINITION 2.4.8 : Let $R = \langle |R|, \preceq, +, \cdot, \mathbf{0}, \mathbf{1}\rangle$ be a grade algebra. We say that R is *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 2.4.3 are bounded usage (Item 1) and distributive lattices (Item 5), while exact usage (Item 1) and linearity (Item 2) are not affine.

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

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

Grade algebras and their homomorphisms form a category denoted by $\mathcal{GrAlg}$, with $\mathcal{GrAlg}^{\mathsf{aff}}$ the full subcategory of the affine grade algebras. We are particularly interested in the initial and final affine grade algebras, since in Chapter 4 they will play a special role.

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

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

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

PROPOSITION 2.4.9 : The following facts hold:

1. $\mathsf{Nat}^{\leq}$ is the initial object in $\mathcal{GrAlg}^{\mathsf{aff}}$;
2. Triv is the terminal object in $\mathcal{GrAlg}^{\mathsf{aff}}$.

Proof: Item 2 is straightforward as the singleton set is a terminal object in the category of sets and functions. Towards a proof of Item 1, let $f\colon \mathsf{Nat}^{\leq} \to R$ be a grade algebra homomorphism and note that, since $n = 1+\cdots+1$ (n times), for all $n \in \mathbb{N}$, and f preserves sums and the unit, we get $f(n) = f(1) +_R \cdots +_R f(1) = \mathbf{1}_R +_R \cdots +_R \mathbf{1}_R$ (n times). That is, we have $f(n) = \iota_R(n)$, for all $n \in \mathbb{N}$. Therefore, to conclude, we just have to show that the map ι_R is a grade algebra homomorphism. The fact that $\iota_R(0) = \mathbf{0}_R$ and $\iota_R(1) = \mathbf{1}_R$ is immediate. The fact that $\iota_R(n+m) = \iota_R(n) +_R \iota_R(m)$ and $\iota_R(n \cdot m) = \iota_R(n) \cdot_R \iota_R(m)$ follows from a straightforward induction on n, using distributivity and nullity properties of the grade algebra R. Finally, to prove monotonicity, consider $n \leq m$ and proceed by induction on $m - n$. If $m - n = 0$, then $n = m$ and so the thesis is trivial. If $m - n = k+1$, we have $m - (n+1) = k$, then by induction hypothesis we get $\iota_R(n+1) \preceq_R (\iota_R(m)$. Since $\iota_R(n+1) = \iota_R(n) +_R \iota_R(1)$ and $\mathbf{0}_R \preceq_R \iota_R(1)$, we get $\iota_R(n) = \iota_R(n) +_R \mathbf{0}_R \preceq_R \iota_R(n) +_R \iota_R(1) \preceq_R \iota_R(m)$, as needed. □

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

Assume a grade algebra $R = \langle |R|, \preceq, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$ and a set X. The set of functions from X to $|R|$ carries a partially ordered commutative monoid structure given by the pointwise extension of the additive structure of R. That is, given $\gamma, \gamma' : X \to |R|$, we define $\gamma \preceq \gamma'$ iff, for all $x \in X$, $\gamma(x) \preceq \gamma'(x)$, and $(\gamma + \gamma')(x) = \gamma(x) + \gamma'(x)$ and $\hat{\mathbf{0}}(x) = \mathbf{0}$, for all $x \in X$. Moreover, we can define a *scalar multiplication*, combining elements of $|R|$ and a function $\gamma : X \to |R|$; indeed, we set $(r \cdot \gamma)(x) = r \cdot \gamma(x)$, for all $r \in |R|$ and $x \in X$. It is easy to see that this operation turns the partially ordered commutative monoid of functions from X to $|R|$ into a partially ordered R-module.

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

In Chapter 3, Chapter 4, and Chapter 5, the type systems rely on *structural coeffect contexts*, that is, (representations of) functions in R^X, with X set of variables. The fact that structural coeffect contexts form a module has been firstly noted in [46, 56]. In Chapter 6 we will show a *non-structural* example. That is, a module different from R^X described above, used mostly in the literature, is needed, where operations on coeffect contexts are not pointwise.

3

Graded Featherweight Java

The main source of inspiration for the work presented in this and the following chapter has been Granule [49], a fully-fledged functional programming language equipped with graded modal types. In Granule, different kinds of grades can be used at the same time; however, available grades are fixed in the language. Our aim is to study a similar support for Java-like languages. To this end, we introduce in a variant of Featherweight Java [39] (FJ for short), the paradigmatic calculus used in literature to model such kind of languages, types decorated with grades, taken, parametrically, *in an arbitrary grade algebra.*

Moreover, we want to prove that the graded type system we propose *overapproximates* the use of resources. Since resource usage is not modeled in standard operational semantics, we also define a *resource-aware semantics* for FJ, parametric on an arbitrary grade algebra as well, which tracks how much each available resource is consumed at each step, and is stuck when the needed amount of a resource is not available. The semantics is given *independently* from the type system, as it is the standard approach in calculi.

We prove a soundness theorem, stating that the graded type system guarantees resource-aware soundness. More technically, typechecking generates annotations in expressions, for annotated expressions produced in this way there is a reduction sequence which does not get stuck.

As already discussed in Section 2.3, we take a much lighter approach with respect to existing literature on graded modal types, likely more convenient for Java-like languages, where the syntax of terms is not affected, and the production for types is $T ::= \tau^r$, that is, all types are (once) graded.

In this and the following chapter, we assume the underlying grade algebra to be *affine*, see Definition 2.4.8. This is due to the fact that, in record/object calculi, generally width subtyping is allowed, meaning that components can be discarded at runtime, whereas object construction happens by sequential evaluation of the fields, see at page 85 for more comments.

In Section 3.1 we report the standard FJ calculus without grades. In Section 3.2 and Section 3.3 we describe the resource-aware semantics, and the resource-aware type system, respectively, providing in Section 3.4 a resource-aware version of soundness with its proof.

3.1 Java-like calculus

The calculus is a variant of FJ [39]. The syntax is reported in the top section of Figure 3.1. We write *es* as metavariable for $e_1, \ldots, e_n$, $n \geq 0$, and analogously for other sequences. We assume *variables* $x, y, z, \ldots$, *class names* C, D, *field names* f, and *method names* m. Types are distinct from class names to mean that they could be extended to include other types, e.g., primitive types, as we will do in Chapter 6. In addition to the standard FJ constructs, we have a block expression, consisting of a local variable declaration, and a body.

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

- $\mathsf{fields}(C)$ gives, for each class C, the sequence $\tau_1\ f_1\texttt{;}\ \ldots\ \tau_n\ f_n\texttt{;}$ of its fields, assumed to have distinct names, with their types[1]
- $\mathsf{mbody}(C, m)$ gives, for each method m of class C, its parameters and body

The semantics is defined differently from the original one; that is, reduction is defined on *configurations* $e|\rho$, where ρ is an *environment*, a finite map from variables into values. In this way, variable occurrences are replaced one at a time by their value in the environment, rather than once and for all.

This definition can be easily shown to be equivalent to the original one, and it is convenient for our aims since, in this presentation, free variables in an expression can be naturally seen as *resources* which are consumed each time a variable occurrence is *used* (replaced by its value) during execution. In other words, this semantics can be naturally *instrumented* by adding grades expressing the "cost" of resource consumption, as we will do in Section 3.2.

Apart from that, the rules in Figure 3.1 are straightforward; only note that, in rules (INVK) and (BLOCK), parameters (including `this`) and local variable are renamed to fresh variables, to avoid clashes. Single contextual rules are given, rather than defining evaluation contexts, to be uniform with the instrumented version, where this presentation is more convenient.

We preferred the more general term *environment* for the map ρ, rather than *heap* as, e.g., done by Choudhury et al. [18]. However, our semantics is still *heap-based*, in the sense that the the map is used as in the state monad: resource annotations could in principle either be decremented or incremented, and the amount of a resource used during a computation is the difference between its input and output grade. Instead, Torczon et al. [54] define an *environment-based* big-step operational semantics, where the resource annotations can only be decremented, and provide an upper bound on the amount required during computation.

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

1 We keep a unique fields function as in the original formulation [39]; however, it should be noted that field types only matter in the type system.

e	::=	$x \mid e.f \mid \texttt{new}\ C(es) \mid e.m(es) \mid \{\tau\ x = e;\ e'\}$	expression
τ	::=	C	type (class name)
v	::=	$\texttt{new}\ C(vs)$	value
ρ	::=	$x_1 \mapsto v_1, \ldots, x_n \mapsto v_n$	environment

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

$$\text{(FIELD-ACCESS)}\ \frac{}{\texttt{new}\ C(v_1, \ldots, v_n).f_i|\rho \to v_i|\rho}\ \begin{array}{l}\mathsf{fields}(C) = \tau_1\ f_1; \ldots \tau_n\ f_n; \\ i \in 1..n\end{array}$$

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

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

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

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

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

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

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

FIGURE 3.1 Syntax and standard reduction

$$\text{(T-SUB)}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'}\ \tau \leq \tau' \qquad \text{(T-VAR)}\ \frac{}{\Gamma \vdash x : \tau}\ \Gamma(x) = \tau$$

$$\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash e : C}{\Gamma \vdash e.f_i : \tau_i}\ \begin{array}{l}\mathsf{fields}(C) = \tau_1\ f_1; \ldots \tau_n\ f_n; \\ i \in 1..n\end{array}$$

$$\text{(T-NEW)}\ \frac{\Gamma \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Gamma \vdash \texttt{new}\ C(e_1, \ldots, e_n) : C}\ \mathsf{fields}(C) = \tau_1\ f_1; \ldots \tau_n\ f_n;$$

$$\text{(T-INVK)}\ \frac{\Gamma \vdash e_0 : C \qquad \Gamma \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Gamma \vdash e_0.m(e_1, \ldots, e_n) : \tau}\ \mathsf{mtype}(C, m) = \tau_1 \ldots \tau_n \to \tau$$

$$\text{(T-BLOCK)}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \{\tau\ x = e;\ e'\} : \tau'}$$

$$\text{(T-ENV)}\ \frac{\Gamma \vdash v_i : \tau_i \quad \forall i \in 1..n}{\Gamma \vdash \rho}\ \begin{array}{l}\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n \\ \rho = x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\end{array}$$

$$\text{(T-CONF)}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \rho}{\Gamma \vdash e|\rho : \tau}$$

FIGURE 3.2 Standard type system

- $\mathsf{mtype}(C, m)$ gives, for each method m of class C, its parameter types and return type
- $\leq$ is the reflexive and transitive closure of the `extends` relation

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

(T-METH)	$\mathsf{mtype}(C, m) = \tau_1 \ldots \tau_n \to \tau$ implies $\mathsf{mbody}(C, m) = \langle x_1 \ldots x_n, e \rangle$ and $\texttt{this} : C, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$
(T-INH-FIELDS)	$C \leq D$ implies $\mathsf{fields}(D)$ is a prefix of $\mathsf{fields}(C)$
(T-INH-METH)	$C \leq D$ and $\mathsf{mtype}(D, m) = \tau_1 \ldots \tau_n \to \tau$ imply $\mathsf{mtype}(C, m) = \tau_1 \ldots \tau_n \to \tau'$ with $\tau' \leq \tau$

Condition (T-METH) expresses that method bodies should conform to method types. Condition (T-INH-FIELDS) expresses that fields are inherited, and, together with the assumption that they have distinct names, that there is no field hiding. Finally, condition (T-INH-METH) expresses that methods are inherited, cannot be overloaded, and can be overriden with a more specific return type.

Typing rules are shown in Figure 3.2 and are standard.

3.2 Resource-aware semantics

This reduction uses *grades*, ranged over by r, s, t, assumed to form a grade algebra, specifying a *partial order* $\preceq$, a *sum* $+$, a *multiplication* $\cdot$, and constants $\mathbf{0}$ and $\mathbf{1}$, satisfying some axioms, as detailed in Definition 2.4.2 of Section 2.4. We additionally assume that the grade algebra is affine, see Definition 2.4.8.

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

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

Before giving the formal definition, we show some simple examples of reductions, considering the (affine variant of the) grade algebra of naturals Example 2.4.3(1), tracking how many times a resource is used.

EXAMPLE 3.2.1 : Assume the following classes:

```
class A {}
class Pair {A left; A right}
```

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

$\texttt{\{A a = [new A()]}_4\texttt{; \{Pair p = [new Pair(a,a)]}_2\texttt{; new Pair(p.left,p.right)\}\}}|\emptyset \to_1$

$\texttt{\{Pair p = [new Pair(a,a)]}_2\texttt{; new Pair(p.left,p.right)\}}|\texttt{a} \mapsto \langle \texttt{new A()}, 4\rangle \to_1$

$\texttt{\{Pair p = [new Pair(new A(),a)]}_2\texttt{; new Pair(p.left,p.right)\}}|\texttt{a} \mapsto \langle \texttt{new A()}, 2\rangle \to_1$

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

$\texttt{new Pair(p.left,p.right)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 2\rangle \to_1$

$\texttt{new Pair(}v_{\texttt{Pair}}\texttt{.left,p.right)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 1\rangle \to_1$

$\texttt{new Pair(new A(),p.right)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 1\rangle \to_1$

$\texttt{new Pair(new A(),}v_{\texttt{Pair}}\texttt{.right)}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 0\rangle \to_1$

$v_{\texttt{Pair}}|\texttt{a} \mapsto \langle \texttt{new A()}, 0\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 0\rangle$

In the example, the top-level reduction is graded 1, meaning that a single value is produced. Subterms are annotated with the grade of their reduction. For instance, the outer initialization expression is annotated 4, meaning that its result can be used (at most) 4 times. To lighten the notation, in this example we omit the index 1. A local variable introduced in a block is added[3] as another

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

3 Modulo renaming to avoid clashes, omitted in the example for simplicity.

available resource in the environment, with the value and the grade of its initialization expression; for instance, the outer local variable is added with grade 4. When evaluating the inner initialization expression, which is reduced with grade 2, each time the variable `a` is used its grade in the environment is decremented by 2.

It is important to notice that the annotations in subterms are *not* type annotations. Except those in arguments of constructor invocation, explained below, annotations are only needed to ensure that reduction of a subterm happens at each step with the same grade, see the formal definition below. In Chapter 5 we will provide an instrumented semantics for a functional calculus which is, instead, in big-step style, hence does not need this artifice. In the example above, we have chosen for the reduction of subterms the minimum grade allowing to perform the top-level reduction. We could have chosen any greater grade; instead, with a strictly lower grade, the reduction would be stuck.

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

EXAMPLE 3.2.2 : Consider the term

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

As highlighted in grey, the first argument of the constructor invocation which is the body of the inner block is now annotated with 2, meaning that the resulting object should have "two copies" of the field. As a consequence, the expression `p.left` should be reduced with grade 2, as shown below, where $v_{\texttt{Pair}} = \texttt{new Pair(new A(),new A())}$, the first four reduction steps are as in Example 3.2.1 and we explicitly write some annotations 1 for clarity

$$\{\texttt{A a} = [\texttt{new A()}]_4\texttt{;}\ \{\texttt{Pair p} = [\texttt{new Pair(}[\texttt{a}]_1\texttt{,a)}]_2\texttt{;}\ \texttt{new Pair(}\ [[\texttt{p}]_1\texttt{.left}]_2\ \texttt{,p.right)}\}\}|\emptyset \rightarrow_1^*$$
$$\texttt{new Pair(}\ [[\texttt{p}]_1\texttt{.left}]_2\ \texttt{,p.right)}|\texttt{a} \mapsto \langle \texttt{new A(),0}\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 2\rangle \rightarrow_1$$
$$\texttt{new Pair(}\ [[v_{\texttt{Pair}}]_1\texttt{.left}]_2\ \texttt{,p.right)}|\texttt{a} \mapsto \langle \texttt{new A(),0}\rangle, \texttt{p} \mapsto \langle v_{\texttt{Pair}}, 1\rangle \qquad \text{STUCK}$$

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

$$\{\texttt{A a} = [\texttt{new A()}]_4\texttt{;}\ \{\texttt{Pair p} = [\texttt{new Pair(}[\texttt{a}]_1\texttt{,a)}]_2\texttt{;}\ \texttt{new Pair(}\ [[\texttt{p}]_2\texttt{.left}]_2\ \texttt{,p.right)}\}\}|\emptyset \rightarrow_1^\star$$

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

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

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

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

The formal definition of the instrumented semantics is given in Figure 3.3. To make the notation lighter, we use the same metavariables of the standard semantics in Figure 3.1. As explained above, reduction is defined on annotated terms. Notably, in each construct, the subterms which are reduced in contextual rules are annotated, so that their reduction always happens with a fixed grade.

In rule (VAR), which is the key rule where resources are consumed, a variable occurrence is replaced by the associated value in the environment, and its grade s decreases to s', burning a non-zero amount r' of resources which has to be at least the reduction grade. The side condition $r' + s' \preceq s$ ensures that the initial grade of the variable suffices to cover both the consumed grade and the residual grade. To show why the amount of resource consumption should be non-zero, consider, e.g., the following variant of Example 3.2.1:

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

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

In rule (FIELD-ACCESS), the reduction grade should be (overapproximated by) the multiplication of the grade of the receiver with that of the field (constructor argument). Indeed, the former specifies how many copies of the object we have and the latter how many copies of the field each of such objects has; thus, their product provides an upper bound to the grade of the resulting value. Note that, in this way, some reductions could be *forbidden*. For instance, taking the (affine) grade algebra of naturals, an access to a field whose value can be used 3 times, of an object reduced with grade 2, can be reduced with grade (at most) 6. Another more significant example is given in the following, taking the grade algebra of *privacy levels*.

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

$$
\begin{array}{lcll}
e & ::= & x \mid [e]_r\,.\,f \mid \texttt{new}\ C\,([e_1]_{r_1}, \ldots, [e_n]_{r_n}) \mid & \text{annotated expression} \\
 & \mid & [e_0]_{r_0}\,.\,m\,([e_1]_{r_1}, \ldots, [e_n]_{r_n}) \mid \{\tau\ x = [e]_r;\ e'\} & \\
v & ::= & \texttt{new}\ C\,([v_1]_{r_1}, \ldots, [v_n]_{r_n}) & \text{annotated value} \\
\rho & ::= & x_1 \mapsto \langle v_1, r_1\rangle, \ldots, x_n \mapsto \langle v_n, r_n\rangle & \text{environment}
\end{array}
$$

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

$$
\textsc{(field-access)}\ \frac{}{[\texttt{new}\ C\,([v_1]_{s_1}, \ldots, [v_n]_{s_n})]_s\,.\,f_i|\rho \to_r v_i|\rho}\ \begin{array}{l} \mathsf{fields}(C) = \tau_1\ f_1;\ \ldots \tau_n\ f_n; \\ i \in 1..n \\ r \preceq s \cdot s_i \end{array}
$$

$$
\textsc{(invk)}\ \frac{}{\begin{array}{l}[v_0]_{s_0}\,.\,m\,([v_1]_{s_1}, \ldots, [v_n]_{s_n})\,|\rho \to_r \\ \quad e[y_0/\texttt{this}][y_1/x_1 \ldots y_n/x_n]|\rho'\end{array}}\ \begin{array}{l} v_0 = \texttt{new}\ C\,(\_) \\ \mathsf{mbody}(C, m) = \langle x_1 \ldots x_n, e\rangle \\ y_0, \ldots, y_n \notin \mathsf{dom}(\rho) \\ \rho' = \rho, y_0 \mapsto \langle v_0, s_0\rangle, \ldots, y_n \mapsto \langle v_n, s_n\rangle \end{array}
$$

$$
\textsc{(block)}\ \frac{}{\{\tau\ x = [v]_s;\ e\}|\rho \to_r e[y/x]|\rho, y \mapsto \langle v, s\rangle}\ y \notin \mathsf{dom}(\rho)
$$

$$
\textsc{(field-access-ctx)}\ \frac{e|\rho \to_s e'|\rho'}{[e]_s\,.\,f|\rho \to_r [e']_s\,.\,f|\rho'}
$$

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

$$
\textsc{(invk-rcv-ctx)}\ \frac{e_0|\rho \to_{r_0} e_0'|\rho'}{[e_0]_{r_0}\,.\,m\,([e_1]_{r_1}, \ldots, [e_n]_{r_n})\,|\rho \to_r [e_0']_{r_0}\,.\,m\,([e_1]_{r_1}, \ldots, [e_n]_{r_n})\,|\rho'}
$$

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

$$
\textsc{(block-ctx)}\ \frac{e_1|\rho \to_s e_1'|\rho'}{\{\tau\ x = [e_1]_s;\ e_2\}|\rho \to_r \{\tau\ x = [e_1']_s;\ e_2\}|\rho'}
$$

FIGURE 3.3 Instrumented (small-step) reduction

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

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

In the grade algebra used in the previous example, grades model *how many times* resources are used. However, grades can also model a non-quantitative[4] information, that is, track possible *modes* in which a resource can be used, or, in other words, possible *constraints* on how it could be used. A typical example of this situation are *privacy levels*, which can be formalized similarly to what is done by Abel and Bernardy [2], as described below.

EXAMPLE 3.2.3 : Starting from any distributive lattice L, like in Example 2.4.3(5), define $\mathsf{L}_0 = \langle |\mathsf{L}_0|, \leq_0, \vee_0, \wedge_0, 0, \top \rangle$, where $|\mathsf{L}_0| = |\mathsf{L}| + \{0\}$ with $0 \leq_0 x$, $x \vee_0 0 = 0 \vee_0 x = x$ and $x \wedge_0 0 = 0 \wedge_0 x = 0$, for all $x \in |\mathsf{L}|$; on elements of $|\mathsf{L}|$ the order and the operations are those of L. That is, we assume that the privacy levels form a distributive semilattice with order representing "decreasing privacy", and we add a grade 0 modeling "non-used". The simplest instance consists of just two privacy levels, that is, $0 \leq \mathsf{priv} \leq \mathsf{pub}$. 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 pub in a subterm, and as priv in another, is overall used as pub. Multiplication is the meet, meaning that we obtain a privacy level which is more restrictive than both: for instance, an access to a field whose value has been obtained in `public` mode, of an object reduced in `private` mode, is reduced in `private` mode[5]. Note that exactly the same structure could be used to model, rather than privacy levels, the modifiers read and mutable in an imperative setting, corresponding to forbid field assignment and no restrictions, respectively. The following examples illustrates the use of this grade algebra. Classes `A` and `Pair` are as in the previous examples.

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

4 These applications are called *informational* by Abel and Bernardy [2].

5 As in *viewpoint adaptation* by Dietl, Drossopoulou, and Müller [28], where permission to a field access can be restricted based on the permission to the base object.

as follows:

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

Instead reduction with grade pub would be stuck since pub $\not\leq$ priv:

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

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

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

would be stuck since $\mathtt{y}|\mathtt{y} \mapsto \langle \mathtt{new\ A()}, \mathsf{priv}\rangle \not\to_{\mathsf{pub}}$. Note that both e_1 and e_2 reduce to `new A()` with the semantics of Figure 3.1.

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

$e_3|\emptyset \to_{\mathsf{pub}} \mathtt{new\ B([x]}_{\mathsf{pub}}\mathtt{, [x]}_{\mathsf{priv}}\mathtt{)}|\mathtt{x} \mapsto \langle \mathtt{new\ A()}, \mathsf{pub}\rangle$ with (BLOCK)
$\to_{\mathsf{pub}} \mathtt{new\ B([new\ A()]}_{\mathsf{pub}}\mathtt{, [x]}_{\mathsf{priv}}\mathtt{)}|\mathtt{x} \mapsto \langle \mathtt{new\ A()}, \mathtt{p\_}\rangle$
with (NEW-CTX) and $\mathtt{x}|\mathtt{x} \mapsto \langle \mathtt{new\ A()}, \mathsf{pub}\rangle \to_{\mathsf{pub}} \mathtt{new\ A()}|\mathtt{x} \mapsto \langle \mathtt{new\ A()}, \mathtt{p\_}\rangle$
$\to_{\mathsf{pub}} \mathtt{new\ B([new\ A()]}_{\mathsf{pub}}\mathtt{, [new\ A()]}_{\mathsf{priv}}\mathtt{)}|\mathtt{x} \mapsto \langle \mathtt{new\ A()}, \mathtt{p\_}\rangle$
with (NEW-CTX) and $\mathtt{x}|\mathtt{x} \mapsto \langle \mathtt{new\ A()}, \mathtt{p\_}\rangle \to_{\mathsf{priv}} \mathtt{new\ A()}|\mathtt{x} \mapsto \langle \mathtt{new\ A()}, \mathtt{p\_}\rangle$

It is easy to see that also

$$e_3|\emptyset \to^*_{\mathsf{priv}} \mathtt{new\ B([new\ A()]}_{\mathsf{pub}}\mathtt{, [new\ A()]}_{\mathsf{priv}}\mathtt{)}|\mathtt{x} \mapsto \langle \mathtt{new\ A()}, \mathtt{p\_}\rangle$$

So we have

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

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

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

However, looking at the reductions of grade pub, only

$$[\mathtt{new\ B([new\ A()]}_{\mathsf{pub}}\mathtt{, [new\ A()]}_{\mathsf{priv}}\mathtt{)}]_{\mathsf{pub}}.\mathtt{left}|\_ \to_{\mathsf{pub}} \mathtt{new\ A()}|\_$$

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

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

PROPOSITION 3.2.4 : If $e|\rho \to_r e'|\rho'$ then $\mathrm{dom}(\rho) \subseteq \mathrm{dom}(\rho')$ and, for all $x \in \mathrm{dom}(\rho)$, $\rho(x) = \langle v, r\rangle$ implies $\rho'(x) = \langle v, s\rangle$ with $s \preceq r$.

PROPOSITION 3.2.5 : If $e|\rho \to_r e'|\rho'$ and $s \leq r$ then $e|\rho \to_s e'|\rho'$.

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

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

PROPOSITION 3.2.6 (Soundness of instrumented semantics): If $e|\rho \to_r e'|\rho'$ then $\lceil e \rceil || \lceil \rho \rceil \to \lceil e' \rceil || \lceil \rho' \rceil$.

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

3.3 Resource-aware type system

Types (class names) are annotated with *grades*, as shown in Figure 3.4.

As anticipated at the end of Section 2.4, a *coeffect context*, of the shape $\gamma = x_1 : r_1, \ldots, x_n : r_n$, where order is immaterial and $x_i \neq x_j$ for $i \neq j$, represents a map from variables to grades (called *coeffects* when used in this position) where only a finite number of variables have non-zero coeffect. A *(type-and-coeffect) context*, of shape $\Gamma = x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n$, with analogous conventions, represents the pair 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 $\{x_1, \ldots, x_n\}$. As customary in type-and-coeffect systems, in typing rules contexts are combined by means of some operations, which are, in turn, defined in terms of the corresponding operations on coeffects (grades). More precisely, we define:

- a partial order $\preceq$

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

- a sum $+$

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

- a scalar multiplication $\cdot$

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

e	::=	$x \mid e.f \mid \texttt{new}\ C(es) \mid e.m(es) \mid$	expression
	$\mid$	$\{T\ x = e;\ e'\}$	
σ, τ	::=	C	non-graded type (class name)
S, T	::=	τ^r	graded type
Γ, Δ	::=	$x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n$	(type-and-coeffect) context

FIGURE 3.4 Syntax with grades, types, and (type-and-coeffect) contexts

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

Type information extracted from the class table is abstractly modeled as at page 16, apart that:

- $\mathsf{fields}(C)$ gives a sequence $\tau_1^{r_1}\ f_1; \ldots \tau_n^{r_n}\ f_n;$, meaning that, to construct an object of type C, we need to provide, for each $i \in 1..n$, a value with a grade at least r_i
- $\mathsf{mtype}(C, m)$ gives, for each method m of class C, its enriched method type, where the types of the parameters and of `this` have grade annotations

The subtyping relation on graded types is defined as follows:

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

That is, a graded type is a subtype of another if the class is a heir class and the grade is more constraining. For instance, taking the affinity grade algebra of Example 2.4.3(2), an invocation of a method with return type τ^∞ can be used in a context where a type τ^1 is required, e.g., to initialize a τ^1 variable.

The conditions on the well-formedness of the class table are adapted as follows:

(T-METH)	$\mathsf{mtype}(C, m) = r_0, \tau_1^{r_1} \ldots \tau_n^{r_n} \to T$ implies $\mathsf{mbody}(C, m) = \langle x_1 \ldots x_n, e \rangle$ and $\texttt{this} :_{r_0} C, x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \vdash e : T$
(T-INH-FIELDS)	$C \leq D$ implies $\mathsf{fields}(D)$ is a prefix of $\mathsf{fields}(C)$
(T-INH-METH)	$C \leq D$ and $\mathsf{mtype}(D, m) = r_0, \tau_1^{r_1} \ldots \tau_n^{r_n} \to T$ imply $\mathsf{mtype}(C, m) = s_0, \tau_1^{s_1} \ldots \tau_n^{s_n} \to S$ with $S \leq T$, $s_i \preceq r_i$ for $i \in 0..n$

In Figure 3.5, we describe the typing rules, which are *parameterized* on the underlying grade algebra.

In rule (T-SUB), both the coeffect context and the (graded) type can be made more general. This means that, on one hand, variables can get less constraining coeffects. For instance, assuming again affinity coeffects, an expression which

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

$$\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash e : C^r}{\Gamma \vdash e.f_i : \tau_i^{r \cdot r_i}}\ \begin{array}{l}\mathsf{fields}(C) = \tau_1^{r_1}\ f_1; \ldots \tau_n^{r_n}\ f_n; \\ i \in 1..n\end{array}$$

$$\text{(T-NEW)}\ \frac{\Gamma_i \vdash e_i : \tau_i^{r \cdot r_i} \quad \forall i \in 1..n}{\Gamma_1 + \ldots + \Gamma_n \vdash \texttt{new}\ C(e_1, \ldots, e_n) : C^r}\ \mathsf{fields}(C) = \tau_1^{r_1}\ f_1; \ldots \tau_n^{r_n}\ f_n;$$

$$\text{(T-INVK)}\ \frac{\Gamma_0 \vdash e_0 : C_0^{r_0} \qquad \Gamma_i \vdash e_i : \tau_i^{r_i} \quad \forall i \in 1..n}{\Gamma_0 + \ldots + \Gamma_n \vdash e_0.m(e_1, \ldots, e_n) : T}\ \mathsf{mtype}(C, m) = r_0, \tau_1^{r_1} \ldots \tau_n^{r_n} \to T$$

$$\text{(T-BLOCK)}\ \frac{\Gamma \vdash e : \tau^r \qquad \Gamma', x :_r \tau \vdash e : T}{\Gamma + \Gamma' \vdash \{\tau^r\ x = e;\ e'\} : T}$$

$$\text{(T-ENV)}\ \frac{\vdash v_i : \tau_i^{r_i} \quad \forall i \in 1..n}{\Gamma \vdash \rho}\ \begin{array}{l}\Gamma = x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \\ \rho = x_1 \mapsto \langle v_1, r_1 \rangle, \ldots, x_n \mapsto \langle v_n, r_n \rangle\end{array}$$

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

FIGURE 3.5 Graded type system

can be typechecked assuming to use a given variable at most once (coeffect 1) can be typechecked as well with no constraints (coeffect ∞). On the other hand, recalling that grades are contravariant in types, an expression can get a more constraining grade. For instance, an expression of grade ∞ can be used where a grade 1 is required.

If we take $r = \mathbf{1}$, then rule (T-VAR) is analogous to the standard rule for variable in coeffect systems, where the coeffect context is the map where the given variable is used once, and no other is used. Here, more generally, the variable can get an arbitrary grade r, provided that it gets the same grade in the context. In other words, as anticipated in Section 2.3, a "local" promotion is applied. However, the use of the variable cannot be just discarded, as expressed by the side condition $r \neq \mathbf{0}$.

In rule (T-FIELD-ACCESS), the grade of the field is multiplied by the grade of the receiver. As already mentioned, this is a form of *viewpoint adaptation* [28].

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

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

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

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

In order to state and prove resource-aware soundness, the typing judgment can be enriched to have shape $\Gamma \vdash e : T \rightsquigarrow e'$, where e' is an annotated expression, as defined in Figure 3.3. That is, typechecking generates annotations in code such that there is a reduction which does not get stuck, as will be formally expressed in the following. This judgment is defined in Figure 3.6.

EXAMPLE 3.3.1 : We show a simple example illustrating the use of graded types, assuming affinity grades. We write in square brackets the grade of the implicit `this` parameter. The class `Pair` declares three versions of the getter for the `left` field, which differ for the grade of the result: either 0, meaning that the result of the method *cannot* be used, or 1, meaning it can be used at

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

$$\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash e : C^r \rightsquigarrow e'}{\Gamma \vdash e.f_i : \tau_i^{r \cdot r_i} \rightsquigarrow [e']_r.f_i}\ \begin{array}{l}\mathsf{fields}(C) = \tau_1^{r_1}\ f_1; \ldots \tau_n^{r_n}\ f_n; \\ i \in 1..n\end{array}$$

$$\text{(T-NEW)}\ \frac{\Gamma_i \vdash e_i : \tau_i^{r \cdot r_i} \rightsquigarrow e_i' \quad \forall i \in 1..n}{\Gamma_1 + \ldots + \Gamma_n \vdash \texttt{new}\ C(e_1, \ldots, e_n) : C^r \rightsquigarrow \texttt{new}\ C([e_1']_{r_1}, \ldots, [e_n']_{r_n})}\ \mathsf{fields}(C) = \tau_1^{r_1}\ f_1; \ldots \tau_n^{r_n}\ f_n;$$

$$\text{(T-INVK)}\ \frac{\begin{array}{c}\Gamma_0 \vdash e_0 : C^{r_0} \rightsquigarrow e_0' \\ \Gamma_i \vdash e_i : \tau_i^{r_i} \rightsquigarrow e_i' \quad \forall i \in 1..n\end{array}}{\Gamma_0 + \ldots + \Gamma_n \vdash e_0.m(e_1, \ldots, e_n) : T \rightsquigarrow [e_0']_{r_0}.m([e_1]_{r_1}, \ldots, [e_n]_{r_n})}\ \mathsf{mtype}(C, m) = r_0, \tau_1^{r_1} \ldots \tau_n^{r_n} \to T$$

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

$$\text{(T-ENV)}\ \frac{\vdash v_i : \tau_i^{r_i} \rightsquigarrow v_i' \quad \forall i \in 1..n}{\Gamma \vdash \rho \rightsquigarrow \rho'}\ \begin{array}{l}\Gamma = x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \\ \rho = x_1 \mapsto \langle v_1, r_1\rangle, \ldots, x_n \mapsto \langle v_n, r_n\rangle \\ \rho' = x_1 \mapsto \langle v_1', r_1\rangle, \ldots, x_n \mapsto \langle v_n', r_n\rangle\end{array}$$

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

FIGURE 3.6 Type system generating annotations

most once, or ∞, meaning it can be used with no constraints. Note that the first version, clearly useless in a functional calculus, could make sense adding effects, e.g. in an imperative calculus, playing a role similar to that of `void`.

```
class Pair { A¹ left; A¹ right;
  A⁰ getLeftZero() [1]{this.left}
  A¹ getLeftAffine() [1]{this.left}
  A^∞ getLeft() [∞]{this.left}
}
```

The coeffect of `this` is 1 in the first two versions, and it is actually used once in the bodies. In the third method, the coeffect needs to be ∞ since the result has the ∞ grade. Fields are graded 1, meaning that a field access does not affect the grade of the receiver.

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

```
Pair^∞ p = ...

{A⁰ a = p.getLeftZero(); new Pair(new A(),new A())}
{A¹ a = p.getLeftAffine(); new Pair(a,new A())}
{A^∞ a = p.getLeft(); new Pair(a,a)}
```

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

```
{A¹ a = p.getLeft(); new Pair(a,a)}
{A^∞ a = p.getLeftAffine(); new Pair(a,a)}
```

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

Finally, note that the coeffect of `this` could be safely changed to be ∞ in the first two methods as well, providing an overapproximated information.

EXAMPLE 3.3.2 : Consider the following source (that is, a non-annotated) version of the expression e_1 of Example 3.2.3.

```
{A^pub y = new A(); {A^priv x = y; x}}
```

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

$$\textsc{(t-block)}\ \frac{\textsc{(t-new)}\ \dfrac{}{\vdash \mathtt{new\ A()} : \mathtt{A}^{\mathsf{pub}}} \qquad \mathcal{D}}{\vdash \{\mathtt{A}^{\mathsf{pub}}\ \mathtt{y} = \mathtt{new\ A()};\ \{\mathtt{A}^{\mathsf{priv}}\ \mathtt{x} = \mathtt{y};\ \mathtt{x}\}\} : \mathtt{A}^{\mathsf{priv}}}$$

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

$$\textsc{(t-block)}\ \frac{\textsc{(t-sub)}\ \dfrac{\textsc{(t-var)}\ \dfrac{}{\mathtt{y} :_{\mathsf{pub}} \mathtt{A} \vdash \mathtt{y} : \mathtt{A}^{\mathsf{pub}}}}{\mathtt{y} :_{\mathsf{pub}} \mathtt{A} \vdash \mathtt{y} : \mathtt{A}^{\mathsf{priv}}} \quad \textsc{(t-var)}\ \dfrac{}{\mathtt{y} :_{\mathsf{pub}} \mathtt{A}, \mathtt{x} :_{\mathsf{priv}} \mathtt{A} \vdash \mathtt{x} : \mathtt{A}^{\mathsf{priv}}}}{\mathtt{y} :_{\mathsf{pub}} \mathtt{A} \vdash \{\mathtt{A}^{\mathsf{priv}}\ \mathtt{x} = \mathtt{y};\ \mathtt{x}\} : \mathtt{A}^{\mathsf{priv}}}$$

On the other hand, initializing a pub variable with a priv expression as in

```
{A^priv y = new A(); {A^pub x = y; x}}
```

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

Consider now the class `B` with a priv field and a pub one.

```
class B { A^pub f1; A^priv f2; }
```

The expression e

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

can be given type $\texttt{B}^{\mathsf{pub}}$ as follows:

$$\dfrac{\dfrac{}{\vdash \texttt{new A()} : \texttt{A}^{\mathsf{pub}}}\,(\textsc{t-new}) \qquad \dfrac{\dfrac{}{\texttt{x} :_{\mathsf{pub}} \texttt{A} \vdash \texttt{x} : \texttt{A}^{\mathsf{pub}}}\,(\textsc{t-var})}{\texttt{x} :_{\mathsf{pub}} \texttt{A} \vdash \texttt{x} : \texttt{A}^{\mathsf{priv}}}\,(\textsc{t-sub}) \qquad \dfrac{}{\texttt{x} :_{\mathsf{pub}} \texttt{A} \vdash \texttt{x} : \texttt{A}^{\mathsf{pub}}}\,(\textsc{t-var})}{\texttt{x} :_{\mathsf{pub}} \texttt{A} \vdash \texttt{new B(x, x)} : \texttt{B}^{\mathsf{pub}}}\,(\textsc{t-new})}{\vdash \{\texttt{A}^{\mathsf{pub}}\ \texttt{x = new A(); new B(x, x)}\} : \texttt{B}^{\mathsf{pub}}}\,(\textsc{t-block})$$

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

$$\dfrac{\vdash e : \texttt{B}^{\mathsf{priv}}}{\vdash e\texttt{.left} : \texttt{A}^{\mathsf{priv}}}\,(\textsc{t-field}) \qquad \dfrac{\vdash e : \texttt{B}^{\mathsf{pub}}}{\vdash e\texttt{.right} : \texttt{A}^{\mathsf{priv}}}\,(\textsc{t-field})$$

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

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

can be given only type $\texttt{B}^{\mathsf{priv}}$ by

$$\dfrac{\dfrac{}{\vdash \texttt{new A()} : \texttt{A}^{\mathsf{priv}}}\,(\textsc{t-new}) \qquad \dfrac{\dfrac{}{\texttt{x} :_{\mathsf{priv}} \texttt{A} \vdash \texttt{x} : \texttt{A}^{\mathsf{priv}}}\,(\textsc{t-var}) \qquad \dfrac{}{\texttt{x} :_{\mathsf{priv}} \texttt{A} \vdash \texttt{x} : \texttt{A}^{\mathsf{priv}}}\,(\textsc{t-var})}{\texttt{x} :_{\mathsf{priv}} \texttt{A} \vdash \texttt{new B(x, x)} : \texttt{B}^{\mathsf{priv}}}\,(\textsc{t-new})}{\vdash \{\texttt{A}^{\mathsf{priv}}\ \texttt{x = new A(); new B(x, x)}\} : \texttt{B}^{\mathsf{priv}}}\,(\textsc{t-block})$$

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

3.4 Resource-aware soundness

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

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

$$\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash_a e : C^r \quad \mathsf{fields}(C) = \tau_1^{r_1}\ f_1\,;\ \ldots\ \tau_n^{r_n}\ f_n\,;}{\Gamma \vdash_a [e]_r\,.\,f_i : \tau_i^{r \cdot r_i}}\ i \in 1..n$$

$$\text{(T-NEW)}\ \frac{\Gamma_i \vdash_a e_i : \tau_i^{r \cdot r_i} \quad \forall i \in 1..n}{\Gamma_1 + \ldots + \Gamma_n \vdash_a \mathtt{new}\ C\,([e_1]_{r_1}, \ldots, [e_n]_{r_n}) : C^r}\ \mathsf{fields}(C) = \tau_1^{r_1}\ f_1\,;\ \ldots\ \tau_n^{r_n}\ f_n\,;$$

$$\text{(T-INVK)}\ \frac{\Gamma_0 \vdash_a e_0 : C^{r_0} \qquad \Gamma_i \vdash_a e_i : \tau_i^{r_i} \quad \forall i \in 1..n}{\Gamma_0 + \ldots + \Gamma_n \vdash_a [e_0]_{r_0}\,.\,m\,([e_1]_{r_1}, \ldots, [e_n]_{r_n}) : T}\ \mathsf{mtype}(C, m) = r_0, \tau_1^{r_1} \ldots \tau_n^{r_n} \to T$$

$$\text{(T-BLOCK)}\ \frac{\Gamma \vdash_a e : \tau^r \qquad \Gamma', x :_r \tau \vdash_a e' : T}{\Gamma + \Gamma' \vdash_a \{\tau\ x = [e]_r\,;\ e'\} : T}$$

$$\text{(T-ENV)}\ \frac{\vdash_a v_i : \tau_i^{r_i} \quad \forall i \in 1..n}{\Gamma \vdash_a \rho}\ \begin{array}{l}\Gamma = x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \\ \rho = x_1 \mapsto \langle v_1, r_1 \rangle, \ldots, x_n \mapsto \langle v_n, r_n \rangle\end{array}$$

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

FIGURE 3.7 Type system for annotated syntax

In order to state and prove a soundness theorem, we need to introduce a (straightforward) typing judgment $\vdash_a$ for annotated expressions, environments and configurations. The typing rules are reported in Figure 3.7.

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

PROPOSITION 3.4.1 : $\Gamma \vdash e : T \rightsquigarrow e'$ if and only if $\lceil e' \rceil = e$ and $\Gamma \vdash_a e' : T$.

A similar property holds for environments and configurations.

The main result is the following progress/subject reduction theorem.

THEOREM 3.4.2 (Progress/Subject reduction): If $\Gamma \vdash_a e|\rho : \tau^r$ then either e is a value or $e|\rho \to_r e'|\rho'$ and $\Gamma' \vdash_a e'|\rho' : \tau^r$ with $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$ and $\Gamma' \preceq \Gamma, \Delta$.

When reduction is non-deterministic, we can distinguish two flavours of soundness, *soundness-must* meaning that no computation can be stuck, and *soundness-may*, meaning that at least one computation is not stuck. The terminology of *may* and *must* properties is very general and comes originally from De Nicola and Hennessy [27]; the specific names soundness-may and soundness-must were introduced in [23, 25] in the context of big-step semantics. In our case, graded reduction is non-deterministic since, as discussed before, the rule (VAR) could be instantiated in different ways, possibly consuming the resource more than necessary. However, we expect that, for a well-typed configuration, there is at least one computation which is not stuck, hence a soundness-may result. Soundness-may can be proved by a theorem like the one above, which

can be seen as a subject-reduction-may result, including standard progress. In our case, if the configuration is well-typed, that is, annotations have been generated by the type system, *there is a step* which leads, in turn, to a well-typed configuration. More in detail, the type is preserved, resources initially available may have reduced grades, and other available resources may be added.

To prove this result, we need some standard lemmas.

LEMMA 3.4.3 (Environment typing): The following facts hold

1. If $\Gamma \vdash_a \rho$ and $\Gamma = \Gamma', x :_r \tau$, then $\rho = \rho', x \mapsto \langle v, r\rangle$ and $\Gamma' \vdash_a \rho'$ and $\vdash v : \tau^r$.
2. If $\Gamma \vdash_a \rho$ and $x \notin \mathrm{dom}(\rho)$ and $\vdash_a v : \tau^r$, then $\Gamma, x :_r \tau \vdash_a \rho, x \mapsto \langle v, r\rangle$.

LEMMA 3.4.4 (Strengthening for values): If $\Gamma \vdash_a v : T$, then $\vdash_a v : T$.

LEMMA 3.4.5 (Canonical forms): If $\Gamma \vdash_a v : \tau^r$ then $v = \mathtt{new}\ D\,([v_1]_{r_1}, \ldots, [v_n]_{r_n})$, $\tau = C$ and $D \leq C$, $\mathrm{fields}(D) = \tau_1^{r_1}\ f_1\mathtt{;}\ \ldots \tau_n^{r_n}\ f_n\mathtt{;}$ and $\Gamma_1 + \ldots + \Gamma_n \preceq \Gamma$ and $\Gamma_i \vdash_a v_i : \tau_i^{s\cdot r_i}$, for all $i \in 1..n$, with $r \preceq s$.

LEMMA 3.4.6 (Renaming): If $\Gamma, x :_r \tau \vdash_a e : T$ and $y \notin \mathrm{dom}(\Gamma)$, then $\Gamma, y :_r \tau \vdash_a e[y/x] : T$.

LEMMA 3.4.7 : If $\mathrm{dom}(\Theta) \cap \mathrm{dom}(\Delta) = \emptyset$ then $(\Gamma, \Theta) + \Delta = (\Gamma + \Delta), \Theta$.

Theorem 3.4.2 is proved as a special case of the following more general result, which makes explicit the invariant needed to carry out the induction. Indeed, by looking at the reduction rules, we can see that computational rules either add new variables to the environment or reduce the grade of a variable of some amount that depends on the grade of the reduction. In the latter case, the amount can be arbitrarily chosen with the only restrictions that it must be non zero and at least the grade of the reduction. However, to prove progress, we not only have to prove that a reduction can be done, but, if the reduction is done in a context, say evaluating the argument of a constructor, then after the reduction we still have enough resources to go on with the reduction, that is, to evaluate the rest of the context (the other arguments of the constructor). This means that the resulting environment has enough resources to type the whole context (the constructor call). For this reason, in the statement of the theorem that follows, we add to the assumption of Theorem 3.4.2 a typing context Θ that would contain the information on the amount of resources that we want to preserve during the reduction (see Item 4 of the theorem). This allows us to choose the appropriate grade to be kept when reducing a variable and to reconstruct a typing derivation when using contextual reduction rules. For the expression at the top level, as we see from the proof of Theorem 3.4.2, Θ is simply $\mathbf{0}$ for all variables in the typing context in which the expression is typed.

THEOREM 3.4.8 : If $\Delta \vdash_a e : \tau^r$ and $\Gamma \vdash_a \rho$ and $\Delta + \Theta \preceq \Gamma$ and $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Theta)$ and e is not a value, then there are e', ρ', Δ', Γ' and Θ' such that

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

Proof: The proof is by induction on typing rules.

(T-VAR) By hypothesis, we know that $\Delta = x :_r \tau$ and, with $r \neq \mathbf{0}$. Since $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Theta)$, we have $\Theta = \Theta_1, x :_{s'} \tau$ and, since $\Delta + \Theta \preceq \Gamma$, we have $\Gamma = \Gamma_1, x :_s \tau$ with $r + s' \preceq s$. Moreover, because $\Gamma \vdash_a \rho$, by Lemma 3.4.3(1), we have that $\rho = \rho', x \mapsto \langle v, s\rangle$, $\Gamma_1 \vdash_a \rho'$ and $\vdash_a v : \tau^s$. Then, by rule (VAR), we get $x|\rho \to_r v|\rho', x \mapsto \langle v, s'\rangle$. Since $s' \preceq r+s' \preceq s$, by rule (T-SUB), we get $\vdash_a v : \sigma^{s'}$ and by Lemma 3.4.3(2) we conclude $\Gamma_1, x :_{s'} \tau \vdash_a \rho', x \mapsto \langle v, s'\rangle$. Since $r \preceq r + s' \preceq s$, again by rule (SUB), we get $\vdash_a v : \tau^r$. Now, let us set $\Delta' = \Theta' = \emptyset$ and $\Gamma' = \Gamma_1, x :_{s'} \tau$. We immediately get $\Delta' \preceq \Delta, \Theta'$ and $\Gamma' \preceq \Gamma, \Theta'$. Furthermore, we get $\Delta' + \Theta \preceq \Gamma'$ as $\Delta' + \Theta = \Theta = \Theta_1, x :_{s'} \tau$ and, from $\Theta \preceq \Delta + \Theta \preceq \Gamma = \Gamma_1, x :_s \tau$, we get $\Theta_1 \preceq \Gamma_1$, therefore $\Theta_1, x :_{s'} \tau \preceq \Gamma_1, x :_{s'} \tau = \Gamma'$, as needed.

(T-SUB) By hypothesis we know that $\Delta_1 \vdash_a e : \tau_1^s$ with $\Delta_1 \preceq \Delta$ and $\tau_1^s \preceq \tau_2^r$, which implies $\tau_1 \preceq \tau_2$ and $r \preceq s$. We distinguish two cases.

- If e is a value then we have the thesis
- Otherwise, notice that $\Delta_1 + \Theta \preceq \Delta\Theta \preceq \Gamma$ holds by monotonicity of +; then, by induction hypothesis, we have
  1. $e|\rho \to_s e'|\rho'$ and
  2. $\Delta' \vdash_a e' : \tau_1^s$ with $\Delta' \preceq \Delta_1, \Theta'$ and
  3. $\Gamma' \vdash_a \rho'$ with $\Gamma' \preceq \Gamma, \Theta'$ and
  4. $\Delta' + \Theta \preceq \Gamma'$.

  By Item 1 and Proposition 3.2.5, since $r \preceq s$, we get $e|\rho \to_r e'|\rho'$. Since $\tau_1^s \preceq \tau_2^r$, from Item 2, by rule (T-SUB), we get $\Delta' \vdash_a e' : \tau_2^r$ and, Since $\Delta_1 \preceq \Delta$, we get $\Delta' \preceq \Delta_1, \Theta' \preceq \Delta, \Theta'$, proving the thesis.

(T-FIELD-ACCESS) By hypothesis we know that $\Delta \vdash_a [e]_r.f_i : \tau_i^{r \cdot r_i}$, $\Delta \vdash_a e_1 : C^r$ and $\mathrm{fields}(C) = \tau_1^{r_1} f_1; \ldots \tau_n^{r_n} f_n;$. We distinguish two cases.

- If e_1 is a value, then, by Lemma 3.4.5, $e_1 = \texttt{new}\ D([v_1]_{s_1}, \ldots, [v_m]_{s_m})$ and $D \preceq C$, $\mathrm{fields}(D) = \sigma_1^{s_1} f_1'; \ldots \sigma_m^{s_m} f_m';$ and $\Delta_1 + \ldots + \Delta_m \preceq \Delta$ and $\Delta_j \vdash_a v_j : \sigma_j^{s \cdot s_j}$, for all $j \in 1..m$, with $r \preceq s$. By coherence conditions on the class table, we know that $n \leq m$ and, for all $j \in 1..n$, $\sigma_j = \tau_j$ and $s_j = r_j$ and $f_j' = f_j$. Hence, since $i \in 1..n$, we have $i \in 1..m$ and so, by rule (FIELD-ACCESS), we get $[e_1]_r.f_i|\rho \to_{r \cdot r_i} v_i|\rho$. Since $r \preceq s$, we have $r \cdot r_i \preceq s \cdot r_i = s \cdot s_i$ and $\tau_i = \sigma_i$, hence, by rule (T-SUB), we derive $\Delta_i \vdash_a v_i : \tau_i^{r \cdot r_i}$. Let us set $\Delta' = \Delta_i$, $\Gamma' = \Gamma$ and

$\Theta' = \emptyset$. Then, the thesis trivially follows as $\Delta' = \Delta_i \preceq \sum_{j=1}^{m} \Delta_j \preceq \Delta$.

- Otherwise, by induction hypothesis, we get
  1. $e_1|\rho \to_r e_1'|\rho'$ and
  2. $\Delta' \vdash_a e_1' : C^r$ with $\Delta' \preceq \Delta, \Theta'$ and
  3. $\Gamma' \vdash_a \rho'$ with $\Gamma' \preceq \Gamma, \Theta'$ and
  4. $\Delta' + \Theta \preceq \Gamma'$.

  Let us set $e' = [e_1']_r \,.\, f_i$. By Item 1 and the hypothesis, using rule (FIELD-ACCESS-CTX), we derive $e|\rho \to_{r \cdot r_i} e'|\rho'$. By Item 2 and the hypothesis, using rule (T-FIELD-ACCESS), we get $\Delta' \vdash_a e' : \tau_i^{r \cdot r_i}$. Finally, by Items 3 and 4, we get the thesis.

(T-NEW) By hypothesis we have $\Delta = \Delta_1 + \ldots + \Delta_n$ and
We distinguish two cases.

- If e_i is a value for all $i \in 1..n$, then e is a value as well and this proves the thesis.
- Otherwise, there is an $i \in 1..n$ such that e_i is not a value while e_j is a value, for all $j \in 1..(i-1)$. Let us set $\widehat{\Theta} = \Theta + \sum_{j=1}^{i-1} \Delta_j + \sum_{j=i+1}^{n} \Delta_j$. Since $\mathsf{dom}(\Delta_i) \subseteq \mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\Theta)$, we have $\mathsf{dom}(\Delta_i) \subseteq \mathsf{dom}(\widehat{\Theta})$ and, by construction, we have $\Delta_i + \widehat{\Theta} = \Delta + \Theta \preceq \Gamma$. Then, by induction hypothesis, we get
  1. $e_i|\rho \to_{r \cdot r_i} e_i'|\rho'$
  2. $\Delta_i' \vdash_a e_i' : \tau_i^{r \cdot r_i}$ with $\Delta_i' \preceq \Delta_i, \Theta'$ and
  3. $\Gamma' \vdash_a \rho'$ with $\Gamma' \preceq \Gamma, \Theta'$ and
  4. $\Delta_i' + \widehat{\Theta} \preceq \Gamma'$.

  Let us set $e' = \texttt{new}\ C\,([v_1]_{r_1}, \ldots, [v_{i-1}]_{r_{i-1}}, [e_i']_{r_i}, \ldots, [e_n]_{r_n})$ and $\Delta' = \sum_{j=1}^{i-1} \Delta_j + \Delta_i' + \sum_{j=i+1}^{n} \Delta_j$. By Item 1 and the hypothesis, using rule (NEW-CTX), we derive $e|\rho \to_r e'|\rho'$. By Item 2 and the hypothesis, using rule (T-NEW), we get $\Delta' \vdash_a e' : C^r$. By Item 3, we have $\mathsf{dom}(\Theta') \cap \mathsf{dom}(\Gamma) = \emptyset$ and, since $\Delta_j \preceq \Delta + \Theta \preceq \Gamma$ for all $j \in 1..n$, we get $\mathsf{dom}(\Delta_j) \cap \mathsf{dom}(\Theta') = \emptyset$. Hence, by monotonicity of + and Lemma 3.4.7, we get

$$\Delta' \preceq \sum_{j=1}^{i-1} \Delta_j + (\Delta_i, \Theta') + \sum_{j=i+1}^{n} \Delta_j = \left(\sum_{j=1}^{n} \Delta_j\right), \Theta' = \Delta, \Theta'$$

  Finally, since $\Delta' + \Theta = \Delta_i' + \widehat{\Theta}$, by Item 4 we get the thesis.

(T-INVK) By hypothesis we know that $[e_0]_{r_0} \,.\, m\,([e_1]_{r_1}, \ldots, [e_n]_{r_n})$, $\Delta = \Delta_0 + \ldots + \Delta_n$, $\mathsf{mtype}(C_0, m) = r_0, \tau_1^{r_1} \ldots \tau_n^{r_n} \to \tau^r$, $\Delta_0 \vdash_a e_0 : C_0^{r_0}$, and $\Delta_i \vdash_a e_i : \tau_i^{r_i}$, for all $i \in 1..n$. Then, we distinguish two cases.

- If, for all $i \in 0..n$, e_i is a value, say $e_i = v_i$, by Lemma 3.4.5, we have $v_0 = \texttt{new}\ D\,(\_)$ with $D \preceq C_0$ and, by coherence conditions

on the class table, we have $\mathsf{mbody}(D, m) = \langle x_1 \dots x_n, e' \rangle$. Then, by rule (INVK), we get $e|\rho \rightarrow_r e'[y_0/\texttt{this}][y_1/x_1 \dots y_n/x_n]|\rho'$ with $y_0, \dots, y_n \notin \mathsf{dom}(\rho)$ and $\rho' = \rho, y_0 \mapsto \langle v_0, r_0 \rangle, \dots, y_n \mapsto \langle v_n, r_n \rangle$. By Lemma 3.4.4, we get $\vdash_a v_i : \tau_i^{r_i}$, for all $i \in 1..n$, hence, by Lemma 3.4.3(2), we get $\Gamma, y_0 :_{r_0} C_0, y_1 :_{r_1} \tau_1 \dots, y_n :_{r_n} \tau_n \vdash_a \rho'$. By condition (T-METH) and rule (T-SUB), we know that $\texttt{this} :_{r_0} C_0$, $x_1 :_{r_1} \tau_1, \dots, x_n :_{r_n} \tau_n \vdash_a e' : C^r$. Let us set $\Delta' = \Theta' = y_0 :_{r_0} C_0, \dots$ $\dots, y_n :_{r_n} \tau_n$, and $\Gamma' = \Gamma, \Theta'$, hence, we immediately have $\Gamma' \preceq \Gamma, \Theta'$ and $\Delta' = \Theta' \preceq \Delta, \Theta'$. By Lemma 3.4.6, we have $\Delta' \vdash_a e'[y_0/\texttt{this}][y_1/x_1 \dots y_n/x_n] : C^r$. Finally, since $\Theta \preceq \Delta + \Theta \preceq \Gamma$ and $y0, \dots, y_n \notin \mathsf{dom}(\Gamma)$ imply $y_0, \dots, y_n \notin \mathsf{dom}(\Theta)$, we get $\Delta' + \Theta = \Theta, \Delta' \preceq (\Delta + \Theta), \Delta' \preceq \Gamma, \Delta' = \Gamma'$.

- Otherwise, there is an $i \in 0..n$ such that e_i is not a value while e_j is a value, for all $j \in 0..(i-1)$. Let us set $\widehat{\Delta} = \sum_{j=0}^{i-1} \Delta_j + \sum_{j=i+1}^{n} \Delta_j$ and $\widehat{\Theta} = \Theta + \widehat{\Delta}$. so $\Delta_i + \widehat{\Theta} = \Delta + \Theta \preceq \Gamma$. By induction hypothesis, we have
  1. $e_i|\rho \rightarrow_{r_i} e_i'|\rho'$
  2. $\Delta_i' \vdash_a e_i' : \tau_i^{r_i}$ where $\tau_0 = C_0$ with $\Delta_i' \preceq \Delta_i, \Theta'$ and
  3. $\Gamma' \vdash_a \rho'$ with $\Gamma' \preceq \Gamma, \Theta'$ and
  4. $\Delta_i' + \widehat{\Theta} \preceq \Gamma'$.

  Let us set $e' = [e_0]_{r_0}.m(\dots, [e_{i-1}]_{r_{i-1}}, [e_i']_{r_i}, [e_{i+1}]_{r_{i+1}}, \dots, [e_n]_{r_n})$ and $\Delta' = \Delta_i' + \widehat{\Delta}$. By Item 1 and the hypothesis, using either rule (INVK-RCV-CTX) or (INVK-ARGS-CTX), depending on whether $i = 0$ or not, we derive $e|\rho \rightarrow_r e'|\rho'$. By Item 2 and the hypothesis, using rule (T-INVK), we have $\Delta' \vdash_a e' : \tau^r$. By Item 3, we have $\mathsf{dom}(\Theta') \cap \mathsf{dom}(\Gamma) = \emptyset$ and, since $\widehat{\Delta} \preceq \Delta + \Theta \preceq \Gamma$, we get $\mathsf{dom}(\widehat{\Delta}) \cap \mathsf{dom}(\Theta') = \emptyset$. Hence, by Item 2, monotonicity of $+$ and Lemma 3.4.7, we get $\Delta' = \Delta_i' + \widehat{\Delta} \preceq (\Delta_i, \Theta') + \widehat{\Delta} = (\Delta_i + \widehat{\Delta}), \Theta' = \Delta, \Theta'$. Finally, since $\Delta' + \Theta = \Delta_i' + \widehat{\Delta} + \Theta = \Delta_i' + \widehat{\Theta}$, by Item 4 we get the thesis.

(T-BLOCK) By hypothesis we have $\Delta = \Delta_1 + \Delta_2$ and $\Delta_1 \vdash_a e_1 : \sigma^s$ and $\Delta_2, x :_s \sigma \vdash_a e_2 : \tau^r$. Then, we distinguish two cases.

- If $e_1 = v$ is a value, then by rule (BLOCK), we have $\{\sigma\, x = [v]_s;\ e_2\}|\rho \rightarrow_r e_2[y/x]|\rho, y \mapsto \langle v, s \rangle$ with $y \notin \mathsf{dom}(\rho)$. By Lemma 3.4.4, we get $\vdash_a v : \sigma^s$, hence, by Lemma 3.4.3(2), we get $\Gamma, y :_D s \vdash_a \rho, y \mapsto \langle v, s \rangle$. Notice that this implies $y \notin \mathsf{dom}(\Gamma)$, thus $y \notin \mathsf{dom}(\Delta_1)$ and $y \notin \mathsf{dom}(\Delta_2)$. Then, let us set $\Theta' = y :_s D$, $\Delta' = \Delta_2, \Theta'$ and $\Gamma' = \Gamma, \Theta'$, hence, we immediately have $\Gamma' \preceq \Gamma, \Theta'$ and $\Delta' = \Delta_2, \Theta' \preceq \Delta, \Theta'$, as $\Delta_2 \preceq \Delta_1 + \Delta_2 = \Delta$. By Lemma 3.4.6, we have $\Delta_2, y :_s \sigma \vdash_a e_2[y/x] : \tau^r$. Finally, since $\Theta \preceq \Gamma$ and $y \notin \mathsf{dom}(\Gamma)$ imply $y \notin \mathsf{dom}(\Theta)$, by Lemma 3.4.7, we get $\Delta' + \Theta = (\Delta_2 + \Theta), \Theta' \preceq (\Delta + \Theta), \Theta' \preceq \Gamma, \Theta' = \Gamma'$.

- Otherwise, let us set $\widehat{\Theta} = \Theta + \Delta_2$. Since $\mathsf{dom}(\Delta_1) \subseteq \mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\Theta)$, we have $\mathsf{dom}(\Delta_1) \subseteq \mathsf{dom}(\widehat{\Theta})$ and, by construction, we also have $\Delta_1 + \widehat{\Theta} = \Delta + \Theta \preceq \Gamma$. By induction hypothesis, we have
  1. $e_1|\rho \to_s e_1'|\rho'$
  2. $\Delta_1' \vdash_a e_1' : \sigma^s$ with $\Delta_1' \preceq \Delta_1, \Theta'$ and
  3. $\Gamma' \vdash_a \rho'$ with $\Gamma' \preceq \Gamma, \Theta'$ and
  4. $\Delta_1' + \widehat{\Theta} \preceq \Gamma'$.

  Let us set $e' = \{\sigma\ x = [e_1']_s;\ e_2\}$ and $\Delta' = \Delta_1' + \Delta_2$. By Item 1 and the hypothesis, using rule (BLOCK-CTX), we derive $e|\rho \to_r e'|\rho'$. By Item 2 and the hypothesis, using rule (T-BLOCK), we have $\Delta' \vdash_a e' : \tau^r$. By Item 3, we have $\mathsf{dom}(\Theta') \cap \mathsf{dom}(\Gamma) = \emptyset$ and, since $\Delta_2 \preceq \Delta + \Theta \preceq \Gamma$, we get $\mathsf{dom}(\Delta_2) \cap \mathsf{dom}(\Theta') = \emptyset$. Hence, by monotonicity of + and Lemma 3.4.7, we get $\Delta' \preceq (\Delta_1, \Theta') + \Delta_2 = (\Delta_1 + \Delta_2), \Theta' = \Delta, \Theta'$. Finally, since $\Delta' + \Theta = \Delta_1' + \widehat{\Theta}$, by Item 4 we get the thesis.

□

We are now ready to prove Theorem 3.4.2.

PROOF OF THEOREM 3.4.2

Proof: Inverting rule (T-CONF), we get $\Gamma \vdash_a \rho$ and $\Delta \vdash_a e : C^r$ with $\Delta \preceq \Gamma$. Applying Theorem 3.4.8 with $\Theta = \mathbf{0} \cdot \Delta$ we get

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

By Item 4, we get $\Delta' \preceq \Gamma'$, hence by (T-CONF) and Items 2 and 3, we conclude $\Gamma' \vdash_a e'|\rho' : C^r$. Finally, by Proposition 3.2.4, we have $\mathsf{dom}(\rho) \subseteq \mathsf{dom}(\rho')$ and, since by rule (T-ENV) and Items 2 and 3 we know that $\mathsf{dom}(\rho) = \mathsf{dom}(\Gamma)$ and $\mathsf{dom}(\rho') = \mathsf{dom}(\Gamma')$, we get the thesis. □

Using Theorem 3.4.2 we can prove a resource-aware soundness theorem. As already noticed, it is a form of soundness-may, that is, it states that a well-typed configuration either converges to a well-typed value or diverges. We write $e|\rho \to_r^\infty$ when there exists an infinite sequence of steps in $\to_r$ starting with $e|\rho$. Note that this judgement can be equivalently defined coinductively by the following rule:

$$\frac{e'|\rho' \to_r^\infty}{e|\rho \to_r^\infty}\ e|\rho \to_r e'|\rho'$$

COROLLARY 3.4.9 (Resource-aware soundness): If $\Gamma \vdash_a e|\rho : C^r$ then either $e|\rho \to_r^\star v|\rho'$ with $\Gamma' \vdash_a v|\rho' : C^r$, or $e|\rho \to_r^\infty$.

Proof: We say that a well-typed configuration $\Gamma \vdash_a e|\rho : C^r$ is well-converging if there are v, ρ', Γ' and Δ such that $e|\rho \to_r^\star v|\rho'$ and $\Gamma' \vdash_a v|\rho' : C^r$ with $\Gamma' \preceq \Gamma, \Delta$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$. The statement is equivalent to the following: if $\Gamma \vdash_a e|\rho : C^r$ and it is not well-converging, then $e|\rho \to_r^\infty$. We prove this by coinduction. Let us consider a well-typed configuration $\Gamma \vdash_a e|\rho : C^r$ which is not well-converging. Then, by Theorem 3.4.2, we get $e|\rho \to_r^\star e'|\rho'$ where $\Gamma' \vdash_a e'|\rho' : C^r$ with $\mathrm{dom}(\Gamma) \subseteq \Gamma'$ and $\Gamma' \preceq \Gamma, \Delta$. To conclude the proof by coinduction, we just have to check that $\Gamma' \vdash_a e'|\rho' : C^r$ is not well-converging. Suppose it is well-converging, then $e'|\rho' \to_r^\star v|\rho''$ where $\Gamma'' \vdash_a v|\rho'' : C^r$ with $\mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma'')$ and $\Gamma'' \preceq \Gamma', \Delta'$. Therefore, we have $e|\rho \to_r e'|\rho' \to_r^\star v|\rho''$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma'')$ and $\Gamma'' \preceq \Gamma', \Delta' \preceq \Gamma, \Delta, \Delta'$, proving that $\Gamma \vdash_a e|\rho : C^r$ is well-converging, which is a contradiction, as needed. □

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

COROLLARY 3.4.10 (Subject reduction/Completeness): If $\Gamma_1 \vdash e_1|\rho_1 : \tau^r \rightsquigarrow e_1'|\rho_1'$ and $e_1|\rho_1 \to e_2|\rho_2$, then $\Gamma_2 \vdash e_2|\rho_2 : \tau^r \rightsquigarrow e_2'|\rho_2'$ with $\mathrm{dom}(\Gamma_1) \subseteq \mathrm{dom}(\Gamma_2)$ and $\Gamma_2 \preceq \Gamma_1, \Delta$, and $e_1'|\rho_1' \to_r e_2'|\rho_2'$.

Proof: By Proposition 3.4.1 we get $\Gamma_1 \vdash_a e_1'|\rho_1' : \tau^r$ and, by Theorem 3.4.2, $e_1'|\rho_1' \to_r e_2'|\rho_2'$ and $\Gamma_2 \vdash_a e_2'|\rho_2' : C^r$ with $\mathrm{dom}(\Gamma_1) \subseteq \mathrm{dom}(\Gamma_2)$ and $\Gamma_2 \preceq \Gamma_1, \Delta$. By Proposition 3.4.1, we get $\Gamma_2 \vdash \lceil e_2' \rceil | \lceil \rho_2' \rceil : \tau^r \rightsquigarrow e_2'|\rho_2'$ and by Proposition 3.2.6, we get $e_1|\rho_1 \to \lceil e_2' \rceil | \lceil \rho_2' \rceil$. By the determinism of the standard semantics we have $\lceil e_2' \rceil = e_2$ and $\lceil \rho_2' \rceil = \rho_2$, hence the thesis. □

4

Multi-graded Featherweight Java

In this chapter, our aim is to make the language *multi-graded*, in the sense that a user could be interested in tracking simultaneously more than one resource usage. For example, both the number of occurences of a variable and its privacy level. This poses the problem of defining the result when grades of different kinds should be combined by the type system. In Section 4.1, we start from two ingredients, a set of *grade kinds* and a family of grade algebras indexed on that, to show a simple way to specify the combination of grades of different kinds by defining a *direct refinement* relation. In Section 4.2 we define a general construction which, given a family of grade algebras and a family of homomorphisms, leads to a unique grade algebra of *heterogeneous grades*. This allows a modular approach, in the sense that the developed meta-theory, including the proof of the results, applies to this case as well. In Section 4.3, finally, we consider the issue of providing linguistic support to specify the desired grade algebras and homomorphisms, hence to make possible for the programmer to define her/his grades.

4.1 Combining grades

As we have seen, each grade algebra encodes a specific notion of resource usage. However, in a program one may need different notions of usage for different kinds of resources or different pieces of code, e.g., different classes. This means that one needs to use several grade algebras at the same time, that is, a family $(H_k)_{k\in\mathcal{K}}$ of grade algebras[1] indexed over a set $\mathcal{K}$ of *grade kinds*. We assume grade kinds to always include N and T, with H_{N} and H_{T} the grade algebras of natural numbers and trivial, respectively. We consider the (affine)[2] grade algebra of natural numbers, as in Example 2.4.3, since they play a special role.

EXAMPLE 4.1.1 : Assume to use, in a program, grade kinds N, A, P, PP, AP, and T, where:

- H_{A} is the affinity grade algebra, as in Example 2.4.3(2).

1 H stands for "heterogeneous".

2 We leave to further work to investigate how the construction presented here should be adapted to non-affine grade algebras.

- H_{P} and H_{PP} are two different instantiations of the grade algebra of privacy levels, as in Example 3.2.3; namely, in H_{P} there are only two privacy levels pub and priv, whereas in H_{PP} we have privacy levels a, b, c, d, with $\mathsf{a} \preceq \mathsf{b} \preceq \mathsf{d}$ and $\mathsf{a} \preceq \mathsf{c} \preceq \mathsf{d}$.
- Finally, H_{AP} is $H_{\mathsf{A}} \times H_{\mathsf{P}}$, as in Example 2.4.3(6), tracking simultaneously affinity and privacy.

We want to make grades of all such kinds simultaneously available to the programmer. In order to achieve this, we should specify how to *combine* grades of different kinds through their distinctive operators; for instance, an object with grade of kind k could have a field with grade of kind μ, hence a field access should be graded by their multiplication.[3]

In other words, we need to construct, starting from the family $(H_k)_{k \in K}$, a single grade algebra of *heterogeneous grades*. In this way, the meta-theory developed in previous sections for an arbitrary grade algebra applies also to the case when several grade algebras are used at the same time. Note that this construction is necessary since we do not want available grades to be fixed, as done by Orchard, Liepelt, and Harley Eades III [49]; rather, the programmer should be allowed to define grades for a specific application, using some linguistic support which could be the language itself, as will be described in Section 4.3.

The obvious approach is to define heterogeneous grades as pairs $\langle k, r \rangle$ where $k \in \mathcal{K}$, and $r \in H_k$. Concerning the definition of the operators, in previous work [8, 9], handling coeffects rather than grades, we took the simplest choice, that is, combining (by either sum or multiplication) grades of different kinds always returns $\langle \infty, \mathsf{T} \rangle$, meaning, in a sense, that we "do not know" how the combination should be done. The only exception are grades of kind N; indeed, since the corresponding grade algebra is initial, we know that, for any kind k, there is a unique grade homomorphism ι_k from $\mathsf{Nat}^{\leq}$ to H_k, hence, to combine $\langle n, \mathsf{N} \rangle$ with $\langle r, k \rangle$, we can map n into a grade of kind k through such homomorphism, and then use the operator of kind k. Here, as done by Bianchini et al. [10], we generalize this idea, by allowing the programmer to specify, for each pair of kinds k and μ, a uniquely determined kind $k \oplus \mu$ and two uniquely determined grade homomorphisms $\mathsf{lh}^H_{\kappa,\mu} \colon H_\kappa \to H_{\kappa \oplus \mu}$, and $\mathsf{rh}^H_{\kappa,\mu} \colon H_\mu \to H_{\kappa \oplus \mu}$. In this way, to combine $\langle \kappa, r \rangle$ and $\langle \mu, s \rangle$, we can map both in grades of kind $k \oplus \mu$, and then use the operator of kind $k \oplus \mu$.

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

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

3 Note that this combination has different aims and properties with respect to the product of grade algebras, as we explain better at the end of this section.

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

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

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

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

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

Altogether, we obtain an instance of a structure called *grade signature*, as will be detailed in Definition 4.2.1. Moreover, given a $\sqsubset^1$-family of homomorphisms:

- they can be extended, by composition[4], to all pairs $\langle \kappa, \mu \rangle \in \mathcal{K} \setminus \{\mathsf{N}, \mathsf{T}\}$ such that there is a path from κ to μ; since this path is unique, the resulting homomorphism is uniquely defined as follows, inductively on the lenght of the path:
  - for all paths κ, $H_{\kappa,\kappa}(r) = r$
  - if $\kappa \sqsubset^1 \nu$ and $\nu \sqsubseteq \mu$ then $H_{\kappa,\mu}(r) = H_{\nu,\mu}(H_{\kappa,\nu}(r))$.
- for each kind κ, we add the unique homomorphisms from $\mathsf{Nat}^{\leq}$ and to Triv.

Besides a grade algebra for each kind, we get a grade homomophism for each pair $\langle \kappa, \mu \rangle$ such that $\kappa \sqsubseteq \mu$. That is, we obtain an instance of a structure called *heterogeneous grade algebra*, as will be detailed in Definition 4.2.2.

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

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

Note the special role played by the grade kinds N and T, with their corresponding grade algebras. The former turns out to be the minimal kind required in a grade signature (Definition 4.2.1); this is important since the zero and one of the resulting grade algebra (hence the zero and one used in the type system) will be those of this kind. The latter, as shown above, is used as default common ancestor for pairs of kinds which do not have one.

4 Note that in this way we obtain, in particular, all the identities.

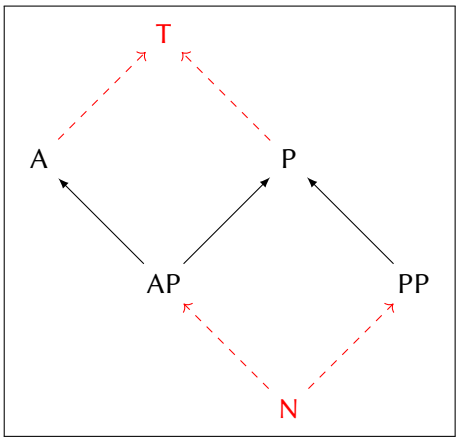

FIGURE 4.1 Direct refinement diagram

EXAMPLE 4.1.2 : Coming back to our example, a programmer could define the direct refinement and the corresponding homomorphisms as follows:

- PP $\sqsubset^1$ P, and the corresponding homomorphism maps, e.g., a and b into priv and c and d into pub
- AP $\sqsubset^1$ A, and AP $\sqsubset^1$ P, and the corresponding homomorphisms are the projections.

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

The direct refinement is pictorially shown in Figure 4.1. Red dotted arrows denote (some of) the order relations added for N and T.

Note that specifying the grade signature and the heterogeneous grade algebra indirectly, by means of the direct refinement and the corresponding homomorphisms, has a fundamental advantage: the semantic check that, for each κ, μ, we can map grades of grade κ into grades of kind μ in a unique way (that is, there is at most one homomorphism), which would require checking the equivalence of function definitions, is replaced by the checks (1) and (2) in the definition of direct refinement, which are purely syntactic and can be easily implemented in a type system (a simple stronger condition is to impose that each kind has a unique parent in the direct refinement, as it is for single inheritance).

In Section 4.3, we will see how to express both grade algebras and homomorphisms as Java classes.

To conclude this section, note that the combination of grades described here has different aim and properties with respect to the product of two grade algebras, as defined in Example 2.4.3(6). Indeed, in the product, the aim is to

simultaneously track the usages tracked by the two components, as shown by H_{AP} in Example 4.1.1; that is, grades in the product carry *more* information than the original ones, and there are obvious homomorphisms (the projections) to the two grade algebras.

In this section, instead, the aim is to combine grades by providing homomorphisms from the original grade algebras to another one, which intuitively carries *less (or equal)* information than both, as, for instance, in Figure 4.1, where both AP and PP are mapped into P. That is, two grades of kind AP and PP, respectively, can be combined by mapping both in grades where we only track privacy levels as in PP. Note also that there is no general way to define a homomorphism from each component grade algebra to the product: for instance, if one of the components is T, the only element can either be mapped into the **0**, or in the **1** element of the product.

4.2 A general construction

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

To develop this construction, we use simple and standard categorical tools, referring to Mac Lane [44] and Riehl [53] for more details. Given a category $\mathcal{C}$, we denote by $\mathcal{C}_0$ the collection of objects in $\mathcal{C}$ and we say that $\mathcal{C}$ is *small* when $\mathcal{C}_0$ is a set. Recall that any partially ordered set $\mathcal{P} = \langle \mathcal{P}_0, \sqsubseteq \rangle$ can be seen as a small category where objects are the elements of $\mathcal{P}_0$ and, for all $x, y \in \mathcal{P}_0$, there is an arrow $x \longrightarrow y$ iff $x \sqsubseteq y$; hence, for every pair of objects in $\mathcal{P}_0$, there is at most one arrow between them, and the only isomorphisms are the identities.

DEFINITION 4.2.1 : A *grade signature* $\mathcal{S}$ is a partially ordered set with finite suprema, that is, it consists of the following data:

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

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

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

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

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

DEFINITION 4.2.2 : A *heterogeneous grade algebra* over the grade signature $\mathcal{S}$ is just a functor $H \colon \mathcal{S} \to \mathcal{GrAlg}^{\text{aff}}$. This means that it consists of a grade algebra $H(\kappa)$, written also H_κ, for every kind $\kappa \in \mathcal{S}_0$, and a grade algebra homomorphism $H_{\kappa,\mu} \colon H_\kappa \to H_\mu$ for every arrow $\kappa \sqsubseteq \mu$, respecting composition and identities[5] , that is, $\kappa \sqsubseteq \mu \sqsubseteq \nu$ implies $H_{\kappa,\nu} = H_{\mu,\nu} \circ H_{\kappa,\mu}$ and $H_{\kappa,\kappa} = \mathsf{id}_{H_\kappa}$.

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

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

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

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

$$\mathsf{lh}^H_{\kappa\oplus\mu,\nu} \circ \mathsf{lh}^H_{\kappa,\mu} = \mathsf{lh}^H_{\kappa,\mu\oplus\nu} \tag{4.1}$$
$$\mathsf{rh}^H_{\kappa\oplus\mu,\nu} \circ \mathsf{lh}^H_{\kappa,\mu} = \mathsf{rh}^H_{\kappa,\mu\oplus\nu} \circ \mathsf{lh}^H_{\mu,\nu} \tag{4.2}$$
$$\mathsf{lh}^H_{\kappa,\mu} = \mathsf{rh}^H_{\mu,\kappa} \tag{4.3}$$
$$\mathsf{lh}^H_{\kappa,\kappa} = \mathsf{id}_{H_\kappa} \tag{4.4}$$
$$\mathsf{lh}^H_{\kappa,I} = \mathsf{id}_{H_\kappa} \tag{4.5}$$
$$\mathsf{rh}^H_{\kappa,I} = \mathsf{in}^H_\kappa \tag{4.6}$$

In the following, we will show how to turn a heterogeneous grade algebra into a single grade algebra. The procedure we will describe is based on a general construction due to Grothendieck [36] defined on indexed categories.

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

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

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

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

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

5 The notation $H_{\kappa,\mu}$ makes sense, because between κ and μ there is at most one arrow.

PROPOSITION 4.2.3 : $\langle |G(H)|, \preceq_H \rangle$ is a partially ordered set.

Proof: We have to prove that $\preceq_H$ is reflexive, transitive and antisymmetric.

Given an element $\langle \kappa, r \rangle \in |G(H)|$, since $\kappa \sqsubseteq \kappa$ and $H_{\kappa,\kappa} = \mathsf{id}_{H_\kappa}$, by functoriality of H, we have

$$H_{\kappa,\kappa}(r) = \mathsf{id}_{H_\kappa}(r) = r$$

hence $\langle \kappa, r \rangle \preceq_H \langle \kappa, r \rangle$, which proves reflexivity.

Given $\langle \kappa, r \rangle \preceq_H \langle \mu, s \rangle \preceq_H \langle \nu, t \rangle$, we know that $\kappa \sqsubseteq \mu \sqsubseteq \nu$ and $H_{\kappa,\mu}(r) \preceq_\mu s$ and $H_{\mu,\nu}(s) \preceq_\nu t$ and, by functoriality of H, $H_{\kappa,\nu} = H_{\mu,\nu} \circ H_{\kappa,\mu}$. Therefore, we get

$$H_{\kappa,\nu}(r) = H_{\mu,\nu}(H_{\kappa,\mu}(r)) \preceq_\nu H_{\mu,\nu}(s) \preceq_\nu t$$

hence $\langle \kappa, r \rangle \preceq_H \langle \nu, t \rangle$, which proves transitivity.

Given $\langle \kappa, r \rangle \preceq_H \langle \mu, s \rangle \preceq_H \langle \kappa, r \rangle$, we know that $\kappa \sqsubseteq \mu \sqsubseteq \kappa$ and $H_{\kappa,\mu}(r) \preceq_\mu s$ and $H_{\mu,\kappa}(s) \preceq_\kappa r$. Since $\sqsubseteq$ is antisymmetric, we get $\kappa = \mu$, hence $H_{\kappa,\mu} = H_{\mu,\kappa} = H_{\kappa,\kappa}$, which, and, by functoriality of H, is equal to the identity id_{H_κ}. This implies that $r \preceq_\kappa s$ and $s \preceq_\kappa r$, which implies $r = s$ by antisymmetry of $\preceq_\kappa$.

□

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

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

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

PROPOSITION 4.2.4 : $\langle |G(H)|, \preceq_H, +_H, \mathbf{0}_H \rangle$ is an ordered commutative monoid.

Proof: We check the four properties.

MONOTONICITY OF $+_H$ Consider $\langle \kappa, r \rangle \preceq_H \langle \mu, s \rangle$ and $\langle \kappa', r' \rangle \preceq_H \langle \mu', s' \rangle$, then we know that $\kappa \sqsubseteq \mu$ and $\kappa' \sqsubseteq \mu'$ and $H_{\kappa,\mu}(r) \preceq_\mu s$ and $H_{\kappa',\mu'}(r') \preceq_{\mu'} s'$. By monotonicity of $\mathsf{lh}^H_{\mu,\mu'}$ and $\mathsf{rh}^H_{\mu,\mu'}$ and $+_{\mu\oplus\mu'}$, we get

$$\mathsf{lh}^H_{\mu,\mu'}(H_{\kappa,\mu}(r)) +_{\mu\oplus\mu'} \mathsf{rh}^H_{\mu,\mu'}(H_{\kappa',\mu'}(r')) \preceq_{\mu\oplus\mu'} \mathsf{lh}^H_{\mu,\mu'}(s) +_{\mu\oplus\mu'} \mathsf{rh}^H_{\mu,\mu'}(s')$$

By monotonicity of $\oplus$, we get $\kappa \oplus \kappa' \sqsubseteq \mu \oplus \mu'$, then, by functoriality of H and by definition of lh^H and $injr$, we have

$$\mathsf{lh}^H_{\mu,\mu'} \circ H_{\kappa,\mu} = H_{\kappa\oplus\kappa',\mu\oplus\mu'} \circ \mathsf{lh}^H_{\kappa,\kappa'} \qquad \mathsf{rh}^H_{\mu,\mu'} \circ H_{\kappa',\mu'} = H_{\kappa\oplus\kappa',\mu\oplus\mu'} \circ \mathsf{rh}^H_{\kappa,\kappa'}$$

therefore, we get

$$\begin{array}{l} H_{\kappa\oplus\kappa',\mu\oplus\mu'}(\mathsf{lh}^H_{\kappa,\kappa'}(r) +_{\kappa\oplus\kappa'} \mathsf{rh}^H_{\kappa,\kappa'}(r')) \preceq_{\mu\oplus\mu'} \\ \mathsf{lh}^H_{\mu,\mu'}(H_{\kappa,\mu}(r)) +_{\mu\oplus\mu'} \mathsf{rh}^H_{\mu,\mu'}(H_{\kappa',\mu'}(r')) \preceq_{\mu\oplus\mu'} \\ \mathsf{lh}^H_{\mu,\mu'}(s) +_{\mu\oplus\mu'} \mathsf{rh}^H_{\mu,\mu'}(s') \end{array}$$

This proves $\langle \kappa, r\rangle +_H \langle \kappa', r'\rangle \preceq_H \langle \mu, s\rangle +_H \langle \mu', s'\rangle$, as needed.

ASSOCIATIVITY OF $+_H$ Consider elements $\langle \kappa, r\rangle$, $\langle \mu, s\rangle$ and $\langle \nu, t\rangle$ in $|G(H)|$. Using Equations (4.1) and (4.2), we have the following

$$\begin{aligned}(\langle \kappa, r\rangle +_H \langle \mu, s\rangle) +_H \langle \nu, t\rangle &= \langle \kappa \oplus \mu, \mathsf{lh}^H_{\kappa,\mu}(r) + \mathsf{rh}^H_{\kappa,\mu}(s)\rangle +_H \langle \nu, t\rangle \\ &= \langle (\kappa \oplus \mu) \oplus \nu, \mathsf{lh}^H_{\kappa\oplus\mu,\nu}(\mathsf{lh}^H_{\kappa,\mu}(r) + \mathsf{rh}^H_{\kappa,\mu}(s)) + \mathsf{rh}^H_{\kappa\oplus\mu,\nu}(t)\rangle \\ &= \langle \kappa \oplus (\mu \oplus \nu), \mathsf{lh}^H_{\kappa,\mu\oplus\nu}(r) + (\mathsf{rh}^H_{\kappa,\mu\oplus\nu}(\mathsf{lh}^H_{\mu,\nu}(s) + \mathsf{rh}^H_{\mu,\nu}(t))\rangle \\ &= \langle \kappa, r\rangle +_H (\langle \mu, s\rangle +_H \langle \nu, t\rangle)\end{aligned}$$

COMMUTATIVITY OF $+_H$ Consider elements $\langle \kappa, r\rangle$ and $\langle \mu, s\rangle$ in $|G(H)|$. Using Equation (4.3), we get the following

$$\begin{aligned}\langle \kappa, r\rangle +_H \langle \mu, s\rangle &= \langle \kappa \oplus \mu, \mathsf{lh}^H_{\kappa,\mu}(r) +_{\kappa\oplus\mu} \mathsf{rh}^H_{\kappa,\mu}(s)\rangle = \langle \kappa \oplus \mu, \mathsf{rh}^H_{\kappa,\mu}(s) +_{\kappa\oplus\mu} \mathsf{lh}^H_{\kappa,\mu}(r)\rangle \\ &= \langle \mu \oplus \kappa, \mathsf{lh}^H_{\mu,\kappa}(s) +_{\mu\oplus\kappa} \mathsf{rh}^H_{\mu,\kappa}(r)\rangle = \langle \mu, s\rangle +_H \langle \kappa, r\rangle\end{aligned}$$

NEUTRALITY OF $\mathbf{0}_H$ Consider $\langle \kappa, r\rangle$ in $|G(H)|$. Using Equations (4.3), (4.5) and (4.6), we the the following

$$\begin{aligned}\langle \kappa, r\rangle +_H \mathbf{0}_H &= \langle \kappa \oplus I, \mathsf{lh}^H_{\kappa,I}(r) +_{\kappa\oplus I} \mathsf{rh}^H_{\kappa,I}(\mathbf{0}_I)\rangle = \langle \kappa, r +_\kappa \mathsf{in}^H_\kappa(\mathbf{0}_I)\rangle \\ &= \langle \kappa, r +_\kappa \mathbf{0}_\kappa\rangle = \langle \kappa, r\rangle\end{aligned}$$

□

PROPOSITION 4.2.5 : $\mathbf{0}_H \preceq_H \langle \kappa, r\rangle$ for every $\langle \kappa, r\rangle \in |G(H)|$.

Proof: Since in^H_κ is a grade algebra homomorphism, we have $\mathsf{in}^H_\kappa(\mathbf{0}_I) = \mathbf{0}_\kappa$ and by definition of grade algebra we have $\mathbf{0}_\kappa \preceq_\kappa r$. Therefore, $\mathsf{in}^H_\kappa(\mathbf{0}_I \preceq_\kappa r$, which proves the thesis by definition of $\preceq_H$. □

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

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

$$\mathbf{1}_H = \langle I, \mathbf{1}_I\rangle$$

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

PROPOSITION 4.2.6 : $\langle |G(H)|, \preceq_H, \cdot_H, \mathbf{1}_H\rangle$ is an ordered monoid.

Proof: To prove the equational axioms of monoid (associativity, and neutrality) the proof is the same as Proposition 4.2.4 when all the involved elements are different from $\mathbf{0}_I$, and it is trivial otherwise. Indeed, if one of such elements is $\mathbf{0}_I$, then the whole multiplication gives $\mathbf{0}_I$ by definition.

To prove monotonicity of $\cdot_H$, consider $\langle \kappa, r\rangle \preceq_H \langle \mu, s\rangle$ and $\langle \kappa', r'\rangle \preceq_H \langle \mu', s'\rangle$ in $|G(H)|$. If they are all different from $\mathbf{0}_I$, the proof goes as in Propos-

ition 4.2.4. If either $\langle \kappa, r\rangle = \mathbf{0}_I$ or $\langle \kappa', r'\rangle = \mathbf{0}_I$, then $\langle \kappa, r\rangle \cdot_H \langle \kappa', r'\rangle = \mathbf{0}_I$ and so the thesis follows by Proposition 4.2.5. If $\langle \mu, s\rangle = \mathbf{0}_I$, then $\langle \kappa, r\rangle \preceq_H \langle \mu, s\rangle = \langle I, \mathbf{0}_I\rangle$ implies $\kappa \sqsubseteq I$ and $H_{\kappa,I}(r) \preceq_I \mathbf{0}_I$ and, since $I \sqsubseteq \kappa$ by definition of grade signature, we get $\kappa = I$. Therefore, by functoriality of H, we have $H_{\kappa,I} = \mathsf{id}_{H_I}$, hence $r \preceq_I \mathbf{0}_I$ which implies $r = \mathbf{0}_I$, since H_I is a grade algebra. This proves that $\langle \kappa, r\rangle = \langle I, \mathbf{0}_I\rangle = \mathbf{0}_H$ and so we get $\langle \kappa, r\rangle \cdot_H \langle \kappa', r'\rangle = \mathbf{0}_H = \langle \mu, s\rangle \cdot_H \langle \mu', s'\rangle$, as needed. Finally, the case $\langle \mu', s'\rangle = \langle I, \mathbf{0}_I\rangle$ is analogous, hence we get the thesis. □

Altogether, we finally get the following result.

THEOREM 4.2.7 : $G(H) = \langle |G(H)|, \preceq_H, +_H, \cdot_H, \mathbf{0}_H, \mathbf{1}_H\rangle$ is a grade algebra.

Proof: By Propositions 4.2.4 and 4.2.6 we have both the additive and multiplicative monoid structures. Proposition 4.2.5 proves that $\mathbf{0}_H$ is the least element of the order $\preceq_H$. The fact that multiplying by $\mathbf{0}_I$ we get again $\mathbf{0}_I$ holds by definition. Hence, it remains to prove that $\cdot_H$ distributes over $+_H$. To this end, consider $\langle \kappa, r\rangle$, $\langle \mu, s\rangle$ and $\langle \nu, t\rangle$ in $|G(H)|$ and assume they are all different from $\mathbf{0}_I$. Using Equations (4.1) to (4.4), we get the following equations:

$$
\begin{aligned}
\mathsf{lh}^H_{\kappa\oplus\mu,\kappa\oplus\nu} \circ \mathsf{lh}^H_{\kappa,\mu} &= \mathsf{lh}^H_{(\kappa\oplus\mu)\oplus\kappa,\nu} \circ \mathsf{lh}^H_{\kappa\oplus\mu,\kappa} \circ \mathsf{lh}^H_{\kappa,\mu} \\
&= \mathsf{lh}^H_{\kappa\oplus(\kappa\oplus\mu),\nu} \circ \mathsf{rh}^H_{\kappa,\kappa\oplus\mu} \circ \mathsf{lh}^H_{\kappa,\mu} \\
&= \mathsf{lh}^H_{(\kappa\oplus\kappa)\oplus\mu,\nu} \circ \mathsf{lh}^H_{\kappa\oplus\kappa,\mu} \circ \mathsf{lh}^H_{\kappa,\kappa} \\
&= \mathsf{lh}^H_{\kappa\oplus\mu,\nu} \circ \mathsf{lh}^H_{\kappa,\mu} = \mathsf{lh}^H_{\kappa,\mu\oplus\nu} \\
\mathsf{rh}^H_{\kappa\oplus\mu,\kappa\oplus\nu} \circ \mathsf{lh}^H_{\kappa,\nu} &= \mathsf{lh}^H_{(\kappa\oplus\mu)\oplus\kappa,\nu} \circ \mathsf{rh}^H_{\kappa\oplus\mu,\kappa} = \mathsf{lh}^H_{\kappa\oplus(\kappa\oplus\mu),\nu} \circ \mathsf{lh}^H_{\kappa,\kappa\oplus\mu} \\
&= \mathsf{lh}^H_{(\kappa\oplus\kappa)\oplus\mu,\nu} \circ \mathsf{lh}^H_{\kappa\oplus\kappa,\mu} \circ \mathsf{lh}^H_{\kappa,\kappa} \\
&= \mathsf{lh}^H_{\kappa\oplus\mu,\nu} \circ \mathsf{lh}^H_{\kappa,\mu} \\
&= \mathsf{lh}^H_{\kappa,\mu\oplus\nu} \\
\mathsf{lh}^H_{\kappa\oplus\mu,\kappa\oplus\nu} \circ \mathsf{rh}^H_{\kappa,\mu} &= \mathsf{lh}^H_{(\kappa\oplus\mu)\oplus\kappa,\nu} \circ \mathsf{lh}^H_{\kappa\oplus\mu,\kappa} \circ \mathsf{rh}^H_{\kappa,\mu} \\
&= \mathsf{lh}^H_{\kappa\oplus(\kappa\oplus\mu),\nu} \circ \mathsf{rh}^H_{\kappa,\kappa\oplus\mu} \circ \mathsf{rh}^H_{\kappa,\mu} = \mathsf{lh}^H_{(\kappa\oplus\kappa)\oplus\mu,\nu} \circ \mathsf{rh}^H_{\kappa\oplus\kappa,\mu} \\
&= \mathsf{rh}^H_{\kappa,\mu\oplus\nu} \circ \mathsf{lh}^H_{\mu,\nu} \\
\mathsf{rh}^H_{\kappa\oplus\mu,\kappa\oplus\nu} \circ \mathsf{rh}^H_{\kappa,\nu} &= \mathsf{rh}^H_{(\kappa\oplus\mu)\oplus\kappa,\nu} = \mathsf{rh}^H_{\kappa\oplus\mu,\nu} \\
&= \mathsf{rh}^H_{\kappa,\mu\oplus\nu} \circ \mathsf{rh}^H_{\mu,\nu}
\end{aligned}
$$

which imply the following

$$
\begin{aligned}
&(\langle \kappa, r\rangle \cdot_H \langle \mu, s\rangle) +_H (\langle \kappa, r\rangle \cdot_H \langle \nu, t\rangle) = \\
&\quad = \langle (\kappa \oplus \mu) \oplus (\kappa \oplus \nu), \mathsf{lh}^H_{\kappa\oplus\mu,\kappa\oplus\nu}(\mathsf{lh}^H_{\kappa,\mu}(r) + \mathsf{rh}^H_{\kappa,\mu}(s)) + \mathsf{rh}^H_{\kappa\oplus\mu,\kappa\oplus\nu}(\mathsf{lh}^H_{\kappa,\nu}(r) + \mathsf{rh}^H_{\kappa,\nu}(t))\rangle \\
&\quad = \langle \kappa \oplus (\mu \oplus \nu), (\mathsf{lh}^H_{\kappa,\mu\oplus\nu}(r) \cdot \mathsf{rh}^H_{\kappa,\mu\oplus\nu}(\mathsf{lh}^H_{\mu,\nu}(s))) + (\mathsf{lh}^H_{\kappa,\mu\oplus\nu}(r) \cdot \mathsf{rh}^H_{\kappa,\mu\oplus\nu}(\mathsf{rh}^H_{\mu,\nu}(t)))\rangle \\
&\quad = \langle \kappa \oplus (\mu \oplus \nu), \mathsf{lh}^H_{\kappa,\mu\oplus\nu}(r) \cdot \mathsf{rh}^H_{\kappa,\mu\oplus\nu}(\mathsf{lh}^H_{\mu,\nu}(s) + \mathsf{rh}^H_{\mu,\nu}(t))\rangle \\
&\quad = \langle \kappa, r\rangle \cdot_H (\langle \mu, s\rangle +_H \langle \nu, t\rangle)
\end{aligned}
$$

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

FIGURE 4.2 Syntax with user-defined grades

which proves distributivity when all the elements are different from $\mathbf{0}_H$.

Now, suppose that $\langle \kappa, r\rangle = \mathbf{0}_H$, then we have $\langle \kappa, r\rangle \cdot_H (\langle \mu, s\rangle +_H \langle \nu, t\rangle) = \langle \kappa, r\rangle \cdot_H \langle \mu, s\rangle = \langle \kappa, r\rangle \cdot_H \langle \nu, t\rangle = \mathbf{0}_H$, hence distributivity trivialy holds. Finally, suppose $\langle \mu, s\rangle = \mathbf{0}_H$ (the case $\langle \nu, t\rangle = \mathbf{0}_H$ is similar), then we have $\langle \mu, s\rangle +_H \langle \nu, t\rangle = \langle \nu, t\rangle$ and $\langle \kappa, r\rangle \cdot_H \langle \mu, s\rangle = \mathbf{0}_H$. Therefore, we get

$$\begin{aligned}\langle \kappa, r\rangle \cdot_H (\langle \mu, s\rangle +_H \langle \nu, t\rangle) &= \langle \kappa, r\rangle \cdot_H \langle \nu, t\rangle = \mathbf{0}_H +_H \langle \kappa, r\rangle \cdot_H \langle \nu, t\rangle \\ &= \langle \kappa, r\rangle \cdot_H \langle \mu, s\rangle +_H \langle \kappa, r\rangle \cdot_H \langle \nu, t\rangle\end{aligned}$$

□

4.3 User-defined grades

We describe now an extension of the calculus supporting user-defined grades. The syntax is reported in Figure 4.2. The differences with the syntax in Figure 3.4 are emphasized in grey. The key feature is that in graded types, occurring in local declarations and method types, the grade is in turn an expression of the language, notably a value. As a metavariable for such values we use $\hat{v}$ rather than v to suggest that they are expected to be values of (a subclass of) a *grade class*, that is, a class implementing a grade algebra, as defined in the following. Moreover, we include a block where the local declaration has a non-graded type, and some other Java constructs useful to write methods of grade classes, notably static methods, booleans and conditional, cast and dynamic typecheck. Moreover, we assume that in the class table there can be *abstract* classes, which may declare abstract methods, and cannot be instantiated. We will write $\neg\mathsf{abs}(C)$ to denote that C is non-abstract.

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

STANDARD CLASS TABLE The first part of the class table is a standard FJ class table, without grade annotations. Classes declared in this class table can be *grade classes*, that is, classes implementing methods corresponding to the ingredients of a grade algebra, or *homomorphism classes*, that is, classes implementing the homomorphisms between grade algebras. Both are described in detail below.

We extend the definition of $\mathsf{mtype}(C, m)$ given on page 18 to express static methods as well. That is, we write

- $\mathsf{mtype}(C, m) = \star, \tau_1 \dots \tau_n \to \tau$ to mean that m is an instance method
- $\mathsf{mtype}(C, m) = \tau_1 \dots \tau_n \to \tau$ to mean that m is a static method

Also the condition on well-typedness of method bodies is adapted to manage abstract and static methods.

(T-METH) $\mathsf{mtype}(C, m) = \tau_1 \dots \tau_n \to \tau \wedge \neg\mathsf{abs}(C)$ implies
$\mathsf{mbody}(C, m) = \langle x_1 \dots x_n, e\rangle$ and
$\texttt{this} : C, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e : \tau$

(T-ST-METH) $\mathsf{mtype}(C, m) = \tau_1 \dots \tau_n \to \tau$ implies
$\mathsf{mbody}(C, m) = \langle x_1 \dots x_n, e\rangle$ and
$x_1 : \tau_1, \dots, x_n : \tau_n \vdash e : \tau$

GRADE CLASSES We write $\mathsf{grade}(C)$ to mean that C is a grade class. In the explicit syntax of the class table used to write examples, we will add a `grade` modifier before `class`.

We assume that, if $\mathsf{grade}(C)$ holds, then:

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

We assume the following predefined classes, implementing the predefined grade algebras H_{N} and H_{T}:

```
abstract grade class Nat {
  abstract boolean leq(Nat x);
  abstract Nat sum(Nat x);
  abstract Nat mult(Nat x);
  static Nat zero(){new Zero()}
  static Nat one(){new Succ(Nat.zero())}
}

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

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

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

Predefined natural numbers are implemented in Peano style; we have a class representing zero and a class representing the successor of a natural number. The operations are the standard ones for Peano numbers. The trivial grade algebra is implemented by a class representing the only element and all the operations return this element, as expected.

HOMOMORPHISM CLASSES We write $\mathsf{homo}(D, C_1, C_2)$ to mean that D is an (assumed to be unique) homomorphism class from C_1 to C_2, assumed to be grade classes. In the explicit syntax of the class table used to write examples, we will add a `homo` modifier before `class`, and use a conventional name `HomoFrom`C_1`To`C_2. We assume that, if $\mathsf{homo}(D, C_1, C_2)$ holds, then:

$$\mathsf{mtype}(D, \mathtt{app}) = C_1 \to C_2$$

This static method takes in input a grade of kind C_1 and returns the corresponding grade of kind C_2. By declaring homomorphism classes, the programmer defines the direct refinement: that is, if $\mathsf{homo}(D, C_1, C_2)$ holds for some D, then $C_1 \sqsubset^1 C_2$. The required conditions on $\sqsubset^1$ can be checked by analysing the class table.

We assume the following predefined homomorphism classes for each grade class C, corresponding to the unique homomorphisms $H_{\mathsf{N},k}$ and $H_{k,\mathsf{T}}$ for each kind k:

```
homo class HomoFromNatToC{
  static C app(Nat x){
    if (x instanceof Zero) C.zero() else
      if (x.pred instanceof Zero) C.one() else
        app(((Succ) x).pred).sum(C.one())
    }
}

homomorphism class HomoFromCToTriv{
  static Triv app(C x){new Triv()}
}
```

In `app` method in class `HomoFromNatToC` we have two base cases: the zero of natural numbers is mapped to the zero of the destination algebra and the successor of zero, that is, the one of natural numbers, is mapped to the one of the destination algebra. For any other natural number, first we recursively apply the homomorphism to its predecessor and then this result is summed with the one of the destination algebra using the sum operation of the destination

algebra. Method `app` in class `HomoFromTrivToC` maps every grade to the only element in H_{T}.

EXAMPLE 4.3.1 : We show a Java implementation of Example 4.1.2. First we define the grade classes. Notably, `Affinity` implements H_{A}, `Privacy` implements H_{P}, `PPrivacy` implements H_{PP}, and `APPair` implements H_{AP}.

```
abstract grade class Affinity {
  abstract boolean leq(Affinity x);
  abstract Affinity sum(Affinity x);
  abstract Affinity mult(Affinity x);
  static Affinity zero(){new AffinityZero()}
  static Affinity one(){new One()}
  }

class AffinityZero extends Affinity {
  boolean leq(Affinity x){true}
  Affinity sum(Affinity x){x}
  Affinity mult(Affinity x){this}
}

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

class Omega extends Affinity {
  boolean leq(Affinity x){x instanceof Omega}
  Affinity sum(Affinity x){this}
  Affinity mult(Affinity x){
    if (x instanceof AffinityZero) new AffinityZero()
    else this
  }
}
```

The grade class `Affinity` is extended by three classes: `AffinityZero`, `One` and `Omega`, representing respectively 0,1, and ∞. Since 0 is smaller or equal than every grade, method `leq` in class `AffinityZero` returns true for all arguments. Since `AffinityZero` is the neutral element of sum, its method `sum` returns always the argument and, since `AffinityZero` is the zero of multiplication, its method `mult` returns always 0. In class `One` method `leq` is false only if the input is 0. In method `sum` we have that 1 summed with 0 is 1 and 1 summed with any other grade is ∞. Also, since 1 is the neutral element of multiplication, its method `mult` returns always the argument. Since ∞ is smaller or equal only of itself, we have that its method `leq` returns true only if the argument is ∞. Moreover, since ∞ summed with any other grade is ∞, its method `sum` returns always ∞. In method `mult`, we have to check whether the argument is 0; in that case the result is 0, otherwise the result is ∞.

```
abstract grade class Privacy {
  abstract boolean leq(Privacy x);
  Privacy mult(Privacy x){if (this.leq(x)) this else x}
  Privacy sum(Privacy x){if (this.leq(x)) x else this}
  static Privacy zero(){new PrivacyZero()}
  static Privacy one(){new Public()}
}

class Private extends Privacy {
  boolean leq(Privacy x){!x instanceof PrivacyZero}
}

class Public extends Privacy {
  boolean leq(Privacy x){x instanceof Public}
}

class PrivacyZero extends Privacy {
  boolean leq(Privacy x){x instanceof PrivacyZero}
}
```

The abstract grade class `Privacy` is extended by three classes representing the three grades. Methods `sum` and `mult` are defined once and for all in the abstract class since they have the same definition in all the grades. In `mult` we take as result the more restrictive, that is, the smaller, privacy level whereas in `sum` we take as result the less restrictive, that is, the bigger, privacy level. In a word, the sum is the join and the multiplication is the meet of the grade algebra. The implementation of `leq` methods is straightforward.

```
abstract grade class PPrivacy {
  abstract boolean leq(PPrivacy x);
  PPrivacy mult(PPrivacy x){
    if (this.leq(x)) this else
      if (x.leq(this)) x else
        new LevelA()
  }
  PPrivacy sum(PPrivacy x){
  if (this.leq(x)) x else
      if (x.leq(this)) this else
        new LevelD()
  }
  static Nat zero(){new PPrivacyZero()}
  static Nat one(){new LevelD()}
}

class PPrivacyZero extends PPrivacy {
  boolean leq(PPrivacy x){true}
}

class LevelA extends PPrivacy {
  boolean leq(PPrivacy x){!(x instanceof PPrivacyZero)}
}
```

```
class LevelB extends PPrivacy {
  boolean leq(PPrivacy x){
     x instanceof LevelB or x instanceof LevelD
  }
}

class LevelC extends PPrivacy {
  boolean leq(PPrivacy x){
     x instanceof LevelC or x instanceof LevelD
  }
}

class LevelD extends PPrivacy {
  boolean leq(PPrivacy x){x instanceof LevelD}
}
```

The abstract grade class `PPrivacy` is extended by five classes, representing the five grades. Also here the methods `sum` and `mult` are defined in the abstract class `PPrivacy`, since they have the same definition for all grades, which is the join and the meet of the grade algebra. In the definition of `mult` we have that if the two levels are comparable, then the result is the smaller one, otherwise we return a since the only incomparable grades are b and c. Similar reasoning applies to `sum`.

```
 grade class APPair {
  Affinity left;
  Privacy right;
  boolean leq(APPair x){
   this.left.leq(x.left) and this.right.leq(x.right)
  }
  APPair mult(APPair x){
    new APPair(this.left.mult(x.left),this.right.mult(x.right))
  }
  APPair sum(APPair x){
    new APPair(this.left.sum(x.left),this.right.sum(x.right))
  }
  static APPair zero(){
    new APPair(Affinity.zero(),Privacy.zero())
  }
  static APPair one(){
    new APPair(Affinity.one(),Privacy.one())
  }
}
```

The grade class `APPair` has two fields representing the two components of the pair; on the left we have an `Affinity` grade, on the right we have a `Privacy` grade. All the methods apply the corresponding operation to the two components separately. The zero and the one of this algebra are, respectively, $\langle 0_A, 0_P \rangle$ and $\langle 1_A, 1_P \rangle$.

We define now the homomorphism classes.

```
homo class HomoAPPairToAffine{
```

```
  Affine app(APPair x){x.left}
}

homo class HomoAPPairToPrivacy{
  Privacy app(APPair x){x.right}
}

homo class HomoPPrivacyToPrivacy{
  Privacy app(PPrivacy x){
    if (x instanceof LevelC) new Public() else
    if (x instanceof levelD) new Public() else new Private()
  }
}
```

Method `app` in class `HomoAPPairToAffine` returns the left component of the pair, that is, the component of type `Affinity`, and analogously in `HomoAPPairToPrivacy`. Method `app` in `HomoPPrivacyToPrivacy` checks whether the input is d; if this is the case, then it returns pub, otherwise, the result is priv.

GRADED CLASS TABLE The second part of the class table has grade annotations which are values. Grade annotations could be generalized to be arbitrary expressions; here we use this simpler assumption to make the presentation lighter. We will write $\vdash_{\mathsf{grade}} \hat{v} : C$ to abbreviate $\emptyset \vdash \hat{v} : C$ and $\mathsf{grade}(C)$, where these are judgments in the standard class table, and $\vdash_{\mathsf{grade}} \hat{v}$ if $\vdash_{\mathsf{grade}} \hat{v} : C$ for some C, that is, $\hat{v}$ is a grade value. Note that, since overloading is not allowed, a grade class cannot be extended by another grade class. Hence, the grade class of each grade value is uniquely determined.

In this class table, we have that the enriched method type, returned by function mtype, and the conditions on the well-formedness of the class table, are defined as at page 26, apart that grades are now values of grade classes, and we must manage abstract and static methods. For the latter, we have to add the following condition:

$$\begin{array}{ll} (\textsc{t-st-meth}) & \mathsf{mtype}(C, m) = \tau_1^{\hat{v}_1} \dots \tau_n^{\hat{v}_n} \to \hat{T} \text{ implies} \\ & \mathsf{mbody}(C, m) = \langle x_1 \dots x_n, e \rangle \text{ and} \\ & x_1 :_{\hat{v}_1} \tau_1, \dots, x_n :_{\hat{v}_n} \tau_n \vdash e : \hat{T} \end{array}$$

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

We explain now in detail how grade classes and grade homomorphism classes declared in the standard class table define families of grade algebras and direct refinements, as described in Section 4.1, so to obtain a grade algebra of heterogeneous grades through the construction formally specified in Section 4.2.

KINDS The kinds are the names of the grade classes, including the predefined `Nat` and `Triv`.

GRADE ALGEBRAS For all kinds C, the elements of the grade algebra H_C are the grade values of type C, and the partial order and the operations in H_C are defined in this way, where $\mathsf{nf}(e)$ denotes the normal form[6] of e:

LEQ $\hat{v}_1 \preceq \hat{v}_2$ if and only if $\mathsf{nf}(\hat{v}_1\texttt{.leq(}\hat{v}_2\texttt{)}) = \texttt{true}$

SUM $\hat{v}_1 + \hat{v}_2 = \mathsf{nf}(\hat{v}_1\texttt{.sum(}\hat{v}_2\texttt{)})$

MULTIPLICATION $\hat{v}_1 \cdot \hat{v}_2 = \mathsf{nf}(\hat{v}_1\texttt{.mul(}\hat{v}_2\texttt{)})$

ZERO $\mathbf{0} = \mathsf{nf}(C\texttt{.zero()})$

ONE $\mathbf{1} = \mathsf{nf}(C\texttt{.one()})$

DIRECT REFINEMENT For all kinds C_1, C_2 we have that $C_1 \sqsubset^1 C_2$ if and only if there exists a class D such that $\mathsf{homo}(D, C_1, C_2)$ holds

GRADE HOMOMORPHISMS For all kinds C_1,C_2 such that $C_1 \sqsubset^1 C_2$, hence $\mathsf{homo}(D, C_1, C_2)$ holds for some (unique) D, for all $\hat{v} \in |H_{C_1}|$, $H_{C_1,C_2}(\hat{v}) = \mathsf{nf}(D\texttt{.app(}\hat{v}\texttt{)})$.

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

1. The standard class table, containing declarations of grade and homomorphism classes, is typechecked by the standard compiler.
2. Code containing grade annotations is typechecked accordingly to the graded type system in Figure 4.3.

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

- Code defining grades and their homomorphisms should be *terminating* (that is, the normal forms of the method calls defining the partial order, the operations and the application of a homomorphism should exist), since, as described above, the second typechecking step requires to *execute* code typechecked in the first step.
- Grade classes should satisfy the required axioms, e.g., the sum derived from `sum` methods should be commutative and associative. The same happens, for instance, in Haskell, when one defines instances of `Functor` or `Monad`.
- The `app` method in homomorphism classes should satify the axioms required to be a homomorphism.

Implementations could use in a parametric way auxiliary tools, notably a termination checker to prevent divergence in methods implementing grades

6 Recall that, with the usual notations and terminology of reduction relations, $\rightarrow^\star$ denotes the transitive and reflexive closure of $\rightarrow$, and e' is a *normal form* of e if $e \rightarrow^\star e'$ and there is no e'' such that $e' \rightarrow e''$. It is easy to see that the FJ reduction relation is deterministic, hence the normal form of e, if any, is unique.

and their homomorphisms, and/or a verifier to ensure that they provide the required properties.

The graded type system used to typecheck the graded class table, reported in Figure 4.3, is essentially the instantiation of the parametric graded type system in Figure 3.2 on the grade algebra H of heterogeneous grades obtained from the user-defined grade and homomorphism classes, as illustrated above. Hence, grades are grade values $\hat{v}$, their partial order and operations are those of H, and the subtyping relation $\leq_H$ follows from the partial order relation on grades. Moreover, the typing rule for the block has an additional side condition, hightlighted in grey, checking that the annotation is actually a grade value. Finally, there are (straightforward) rules for the additional constructs, highlighted in grey.

$$\text{(T-SUB)}\ \frac{\Gamma \vdash e : T \quad \Gamma \preceq_H \Gamma' \quad T \preceq_H T'}{\Gamma' \vdash e : T'} \qquad \text{(T-VAR)}\ \frac{}{x :_\tau \hat{v} \vdash x : \tau^{\hat{v}}}\ \hat{v} \neq \mathbf{0}_H$$

$$\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash e : C^{\hat{v}} \quad \mathsf{fields}(C) = \tau_1^{\hat{v}_1}\, f_1; \ldots \tau_n^{\hat{v}_n}\, f_n;}{\Gamma \vdash e.f_i : \tau_i^{\hat{v} \cdot_H \hat{v}_i}}\ i \in 1..n$$

$$\text{(T-NEW)}\ \frac{\Gamma_i \vdash e_i : \tau_i^{\hat{v} \cdot_H \hat{v}_i} \quad \forall i \in 1..n}{\Gamma_1 +_H \ldots +_H \Gamma_n \vdash \texttt{new}\ C(e_1, \ldots, e_n) : C^{\hat{v}}}\ \mathsf{fields}(C) = \tau_1^{\hat{v}_1}\, f_1; \ldots \tau_n^{\hat{v}_n}\, f_n;$$

$$\text{(T-INVK)}\ \frac{\Gamma_0 \vdash e_0 : C^{\hat{v}_0} \qquad \Gamma_i \vdash e_i : \tau_i^{\hat{v}_i} \quad \forall i \in 1..n}{\Gamma_0 +_H \ldots +_H \Gamma_n \vdash e_0.m(e_1, \ldots, e_n) : \tau^{\hat{v}}}\ \mathsf{mtype}(C, m) = \hat{v}_0, \tau_1^{\hat{v}_1} \ldots \tau_n^{\hat{v}_n} \to \tau^{\hat{v}}$$

$$\text{(T-ST-INVK)}\ \frac{\Gamma_i \vdash e_i : \tau_i^{\hat{v}_i} \quad \forall i \in 1..n}{\Gamma_0 +_H \ldots +_H \Gamma_n \vdash C.m(e_1, \ldots, e_n) :\ \tau^{\hat{v}}}\ \mathsf{mtype}(C, m) = \tau_1^{\hat{v}_1} \ldots \tau_n^{\hat{v}_n} \to\ \tau^{\hat{v}}$$

$$\text{(T-BLOCK)}\ \frac{\Gamma \vdash e : \tau^{\hat{v}} \qquad \Gamma', x :_\tau \hat{v} \vdash e' : T}{\Gamma +_H \Gamma' \vdash \{\tau^{\hat{v}}\ x = e;\ e'\} : T}\ \vdash_{\mathsf{grade}} \hat{v}$$

$$\text{(T-IF)}\ \frac{\Gamma \vdash e : \texttt{boolean}^{\hat{v}} \qquad \Delta \vdash e_1 : T \qquad \Delta \vdash e_2 : T}{\Gamma +_H \Delta \vdash \texttt{if}\,(e)\, e_1\, \texttt{else}\ e_2 : T}$$

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

$$\text{(T-TRUE)}\ \frac{}{\vdash \texttt{true} : \texttt{boolean}^{\hat{v}}} \qquad \text{(T-FALSE)}\ \frac{}{\vdash \texttt{false} : \texttt{boolean}^{\hat{v}}}$$

$$\text{(T-ENV)}\ \frac{\vdash_a \hat{v}_i : \tau_i^{r_i} \quad \forall i \in 1..n}{\Gamma \vdash_a \rho}\ \begin{array}{l}\Gamma = x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \\ \rho = x_1 \mapsto \langle \hat{v}_1, r_1 \rangle, \ldots, x_n \mapsto \langle \hat{v}_n, r_n \rangle\end{array}$$

$$\text{(T-CONF)}\ \frac{\Delta \vdash e : \hat{T} \qquad \Gamma \vdash \rho}{\Gamma \vdash e|\rho : \hat{T}}\ \Delta \preceq_H \Gamma$$

FIGURE 4.3 Graded type system with user-defined grades

5

Beyond object-oriented and small-steps

The previous chapters cover the main focus of the thesis, that is, the design of a resource-aware extension for Java-like languages. To this end we provided, for a paradigmatic Java-like calculus, a resource-aware small-step semantics and a graded type-system, both parametric on an arbitrary grade algebra. We proved resource-aware soundness, and, moreover, that the language can be multigraded and grades can be user-defined. In this chapter we go beyond this in two directions.

On the language side, here we consider an extended lambda calculus, intended to be representative of typical features of functional programming languages. This poses additional challenges with respect to the object-oriented case, such as higher-order functions and structural types, including recursive ones. As already mentioned, most literature on graded type systems considers similar lambda calculi, however here, besides the fact that we do not add ad-hoc syntax, as discussed in Section 2.3, 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. Moreover, our graded type system smoothly includes *equi-recursive* types, a feature which once again permits no syntax overhead.

On the foundational side, a significant difference is that the resource-aware semantics is given here in *big-step* style, so that *no annotations are needed.* This is again to follow the "no ad-hoc changes" principle mentioned in Section 2.3, at a more technical level. Indeed, this is unimportant for the standard programmer, but allows a cleaner and simpler way to analyse the behaviour of programs, notably reasoning directly on source code. 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, as shown by Cousot and Cousot [21] and Leroy and Grall [42]. To solve this problem, the big-step judgment is extended to model divergence explicitly, and is defined by a *generalized inference system*, [3, 24]. Another technical improvement is that here we consider a more general class of grade algebras, including *non-affine* grade algebras. So we can use,

e	$::=$	$x \mid \texttt{rec}\, f.\lambda x.e \mid e_1 e_2 \mid$	expression
	$\mid$	$\texttt{unit} \mid \texttt{match}\ e_1\ \texttt{with}\ \texttt{unit} \to e_2$	
	$\mid$	$\langle {}^r e_1, e_2 {}^s \rangle \mid \texttt{match}\ e_1\ \texttt{with}\ \langle x, y \rangle \to e_2 \mid$	
	$\mid$	$\texttt{inl}^r e \mid \texttt{inr}^r e \mid$	
	$\mid$	$\texttt{match}\ e\ \texttt{with}\ \texttt{inl}\ x_1 \to e_1\ \texttt{or}\ \texttt{inr}\ x_2 \to e_2$	
$\mathbf{v}$	$::=$	$\texttt{rec}\, f.\lambda x.e \mid \texttt{unit} \mid \langle {}^r \mathbf{v}_1, \mathbf{v}_2 {}^s \rangle \mid \texttt{inl}^r \mathbf{v} \mid \texttt{inr}^r \mathbf{v}$	value
ρ	$::=$	$x_1 \mapsto \langle \mathbf{v}_1, r_1 \rangle, \ldots, x_n \mapsto \langle \mathbf{v}_n, r_n \rangle$	environment

FIGURE 5.1 Syntax

for instance, the natural numbers with exact usage and the linearity grade algebras.

In Section 5.1 and Section 5.2 we present the resource-aware reduction, and the type system, respectively. In Section 5.3 we prove resource-aware soundness. We provide examples and discussions in Section 5.4.

In this chapter, we assume an integral grade algebra, see Definition 2.4.4. Requiring R to be integral allows us to do some simplifications. In particular, the fact that multiplying non-zero grades we cannot get $\mathbf{0}$ is used, e.g., in the proof of Lemma 5.2.6(2).

5.1 Functional calculus and resource-aware semantics

We define, for a standard functional calculus, an instrumented semantics which, analogously to that defined in Section 3.2 for the Java-like calculus, keeps track of resource usage, hence, in particular, gets stuck if some needed resource is insufficient. However, in this case the semantics will be given in big-step style.

SURFACE SYNTAX The (surface) syntax is given in Figure 5.1. We assume variables $x, y, f, \ldots$, 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 the role of destructors of units, pairs, and injections, respectively. Instead of standard lambda expressions and a `fix` operator for recursion, we have a unique construct $\texttt{rec}\, f.\lambda x.e$, 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 this 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 the 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 Figure 5.5.

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 2.4.3(1), a pair of shape $\langle {}^2e_1, e_2{}^2\rangle$ 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. This is analogous to annotations of a constructor invocation in the FJ calculus, specifying how many copies of each field should contain the object to be constructed. It is not mandatory to have such annotations; we include them to have an additional language feature, that is, to be able to express "data containers" whose subcomponents are graded. In Chapter 3, instead, annotations also play a different role, that is, are needed for all language operators to express the semantics in small-step style.

We will sometimes use, rather than `match` e_1 `with unit` $\to$ 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 As in Section 3.2, the resource-aware semantics is defined on *configurations*, that is, pairs $e|\rho$ where ρ is an *environment* keeping track of the existing resources, that is, as shown in Figure 5.1, 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[1] $\mathbf{v}$ and a final environment ρ'. As in the small-step case, the reduction relation is *graded*, that is, indexed by a grade r, the grade of a variable in the environment decreases each time the variable is used, and, if this is not possible, evaluation is stuck. However, in the big-step case, this is formalized by the fact that no judgment can be derived.

The choice of big-step style is motivated since small-step style, as shown in Section 3.2, needs a syntax where *all* constructs have a grade annotation for each subterm, to ensure that all reduction steps have the same grade. Here, instead, as said above, only constructs 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* [43] version of expressions. However, first we illustrate its expected behaviour on some simple examples in the surface syntax.

EXAMPLE 5.1.1 : Let us consider the following expressions:

1 Here we use the meta-variable $\mathbf{v}$ for values which are results, keeping v for *value expressions*, as explained in the following.

$\rho = \mathtt{p} \mapsto \langle \mathbf{v}, 1\rangle$ with $\mathbf{v} = \langle \mathbf{v}_1, \mathbf{v}_2\rangle$
$\rho' = \mathtt{p} \mapsto \langle \mathbf{v}, 0\rangle, \mathtt{x} \mapsto \langle \mathbf{v}_1, 1\rangle, \mathtt{y} \mapsto \langle \mathbf{v}_2, 1\rangle$
$\rho'' = \mathtt{p} \mapsto \langle \mathbf{v}, 0\rangle, \mathtt{x} \mapsto \langle \mathbf{v}_1, 0\rangle, \mathtt{y} \mapsto \langle \mathbf{v}_2, 1\rangle$

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

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

FIGURE 5.2 Examples of resource-aware evaluation (counting usages)

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

to be evaluated in the environment $\rho = \mathtt{p} : \langle \mathbf{v}, 1\rangle$ with $\mathbf{v} = \langle \mathbf{v}_1, \mathbf{v}_2\rangle$. Assume, first, that grades are natural numbers, see Example 2.4.3(1). In order to lighten the notation, 1 annotations are considered default, hence omitted. In the first proof tree in Figure 5.2 we show the evaluation of e_1. The resource $\mathtt{p}$ is consumed, and its available amount (1) is "transferred" to both the resources $\mathtt{x}$ and $\mathtt{y}$, which are added in the environment[2], and not consumed.

The evaluation of e_2 is similar, apart that the resource $\mathtt{x}$ 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 $\mathtt{x}$ is found, the resource is exhausted, as shown in the second (incomplete) proof tree in Figure 5.2. A result could be obtained, instead, if the original grade of $\mathtt{p}$ was greater than 1 (e.g., 2), since in this case $\mathtt{x}$ (and $\mathtt{y}$) would be added with grade 2, or, alternatively, if the value associated to $\mathtt{p}$ in the environment was, e.g., $\mathbf{v} = \langle {}^2\mathbf{v}_1, \mathbf{v}_2\rangle$.

EXAMPLE 5.1.2 : Assume now that grades are privacy levels $0 \preceq \mathsf{priv} \preceq \mathsf{pub}$ as in Example 3.2.3. We have, e.g., for $\rho = \mathtt{p} : \langle \mathbf{v}, \mathsf{pub}\rangle$ with $\mathbf{v} = \langle {}^{\mathsf{priv}}\mathbf{v}_1, \mathbf{v}_2\rangle$, that the evaluation in mode pub of e_1 is analogous to that in Figure 5.2; however, the evaluation in mode pub of e_2, e_3, and e_4 is stuck, since it needs to use the resource $\mathtt{x}$, which gets a grade $\mathsf{priv} = \mathsf{priv} \cdot \mathsf{pub}$, hence cannot be used in mode pub since $\mathsf{pub} \not\preceq \mathsf{priv}$, as we show in the first (incomplete) proof tree for e_2 in Figure 5.3. On the other hand, evaluation in mode priv can be safely performed; indeed, resource $\mathtt{p}$ can be used in mode priv since $\mathsf{priv} \preceq \mathsf{pub}$, as shown in the second proof tree in Figure 5.3.

2 Modulo renaming, omitted here for simplicity.

$\rho = \mathtt{p} \mapsto \langle \mathbf{v}, \mathsf{pub} \rangle$ with $\mathbf{v} = \langle^{\mathsf{priv}} \mathbf{v}_1, \mathbf{v}_2 \rangle$
$\rho' = \mathtt{p} \mapsto \langle \mathbf{v}, \mathsf{pub} \rangle, \mathtt{x} \mapsto \langle \mathbf{v}_1, \mathsf{priv} \rangle, \mathtt{y} \mapsto \langle \mathbf{v}_2, \mathsf{pub} \rangle$

$$\dfrac{\dfrac{}{\mathtt{p}|\rho \Rightarrow \mathbf{v}|\mathtt{p}:\langle \mathbf{v}, \mathsf{pub} \rangle} \qquad \dfrac{\dfrac{}{\mathtt{x}|\rho' \Rightarrow ?}}{\langle \mathtt{x}, \mathtt{unit} \rangle|\rho' \Rightarrow ?}}{\mathtt{match}\ \mathtt{p}\ \mathtt{with}\ \langle \mathtt{x}, \mathtt{y} \rangle \rightarrow \langle \mathtt{x}, \mathtt{unit} \rangle|\rho \Rightarrow ?}$$

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

FIGURE 5.3 Examples of resource-aware evaluation (privacy levels)

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

FIGURE 5.4 Fine-grained syntax

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 [43] 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*[3]. 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 Figure 5.4. As said above, there are two distinct syntactic categories of value expressions 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

3 They are often called just "values" in literature, though, as already noted by Levy, Power, and Thielecke [43], 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.

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 Figure 5.5. 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, and $c \Rightarrow_r \mathbf{v}|\rho$ in the bottom section, with the latter defined on top of the former. Hence, the metarules in Figure 5.5 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 Figure 5.5 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, (VAR), the key rule where resources are consumed, is analogous to the small-step version in Figure 3.3: a variable is replaced by its associated value, its grade s decreases to s', burning an amount r' of resource which has to be at least the reduction grade, and the side condition $r' + s' \preceq s$ ensures that the initial grade allows to consume the r' amount, leaving a residual grade s'. Differently from rule (VAR) in Figure 3.3, here r can be zero. Indeed, the constraint that consumption of a resource should be non-zero only holds when value expressions are actually "used", see below.

As already noted, 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. By looking at this rule it is worth to distinguish two interpretations of the concept of "waste". The first interpretation is that to waste means to consume more than imposed by the grade on the arrow. Since this kind of wasting depends on the order, it is possible to avoid it by using an algebra with, as order, the equality; in this way r' is equal to r and so we consume exactly the amount imposed by the arrow. The second interpretation is that to waste means to consume all the resources we have in the environment. This kind of waste is not avoidable in this system, since we do not impose any constraint on the remaining resources. In the future work we plan to investigate how to manage this kind of waste, as discussed more in details in Chapter 8.

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 reduction grade of the compound value expression, multiplied by that of the component. They are analogous to

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

$$\text{(FUN)}\ \frac{}{\mathtt{rec}\, f.\lambda x.e|\rho \Rightarrow_r \mathtt{rec}\, f.\lambda x.e|\rho}$$

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

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

$$\text{(RET)}\ \frac{v|\rho \Rightarrow_s \mathbf{v}|\rho'}{\mathtt{return}\ v|\rho \Rightarrow_r \mathbf{v}|\rho'}\ r \preceq s \neq \mathbf{0}$$

$$\text{(LET)}\ \frac{\begin{array}{l} e_1|\rho \Rightarrow_s \mathbf{v}|\rho'' \\ e_2[x'/x]|\rho'', x' \mapsto \langle \mathbf{v}, s\rangle \Rightarrow_r \mathbf{v}'|\rho' \end{array}}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2|\rho \Rightarrow_r \mathbf{v}'|\rho'}\ x' \text{ fresh}$$

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

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

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

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

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

FIGURE 5.5 Instrumented (big-step) reduction

rule (NEW-CTX) in Figure 3.3, where we reduce the subterms of a constructor invocation with a grade that is the multiplication of the reduction grade of the object to be constructed with the grade of the field.

Whereas evaluation of value expressions may have grade **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 Figure 5.5.

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 **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, **1** would not work for grade algebras where there are grades between **0** and **1**, as it happens, e.g., for privacy levels. Note also that t is non-deterministically guessed. 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. We can notice that, differently from some coeffect systems, here we do not include coeffects on pattern matching, but we have the same result by reducing the value to be matched with the desired reduction grade since this reduction grade is free. 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.

$$\dfrac{\dfrac{}{y|\langle r_0,s_0,\mathbf{1}\rangle \Rightarrow \mathtt{u}|\langle r_1,s_0,\mathbf{1}\rangle} \qquad \dfrac{\dfrac{}{f|\langle r_1,s_0,\mathbf{1}\rangle \Rightarrow_{s_0} \mathsf{div}|\langle r_1,\mathbf{0},\mathbf{1}\rangle} \quad \dfrac{}{x|\langle r_1,\mathbf{0},\mathbf{1}\rangle \Rightarrow \mathtt{u}|\langle r_1,\mathbf{0},\mathbf{0}\rangle} \quad \dfrac{\cdots}{y;fx|\langle r_1,s_1,\mathbf{1}\rangle \Rightarrow ?}\,(\textsc{app})}{fx|\langle r_1,s_0,\mathbf{1}\rangle \Rightarrow ?}\,(\textsc{app})}{y;fx|\langle r_0,s_0,\mathbf{1}\rangle \Rightarrow ?}\,(\textsc{match-u})$$

FIGURE 5.6 Example of resource-aware evaluation: consumption/divergency

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.

EXAMPLE 5.1.3 : 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 Figure 5.6 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 $\langle r,s,t\rangle$ the environment $y \mapsto \langle \mathtt{unit}, r\rangle, f \mapsto \langle \mathsf{div}, s\rangle, x \mapsto \langle \mathtt{unit}, t\rangle$, and we omit the label in applications of meta-rule (VAR).

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 2.4.3(1), it is easy to see that the above sequence of constraints can be equivalently expressed as:

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

σ, τ	::=	$T \to_s S \mid \texttt{Unit} \mid T \otimes S \mid T + S$	non-graded type
S, T	::=	τ^r	graded type
Γ, Δ	::=	$x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n$	(type-and-coeffect) context

FIGURE 5.7 Types and (type-and-coeffect) contexts

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 2.4.3(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 $r_k = \mathsf{pub}$ for all k. However, interpreting the rules in Figure 5.5 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 5.3 how to extend the semantics to model non-terminating computations as well.

5.2 Resource-aware type system

Types, defined in Figure 5.7, 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[4] 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.

(Type-and-coeffect) contexts, and their partial order and operations, are defined as in Section 3.3:

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

The partial order is obtained by lifting the corresponding operations on coeffect contexts, which are the pointwise extension of those on coeffects. The difference with respect to the definition of Section 3.3 is that in the third clause there is an additional condition: if a variable is present only in the right context, its grade should be greater or equal than zero. Indeed, if the variable is not present on the left, then its coffect is implicitly assumed to be zero. In the affine grade algebras we consider in Chapter 3 and Chapter 4, this condition

4 More precisely, *regular* terms, that is, those with finitely many distinct subterms.

$$\text{(T-SUB-V)}\ \frac{\Gamma \vdash v : \tau^{r} \quad \Gamma \preceq \Gamma' \quad s \preceq r}{\Gamma' \vdash v : \tau^{s}} \qquad \text{(T-VAR)}\ \frac{}{x :_{r} \tau \vdash x : \tau^{r}}$$

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

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

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

$$\text{(T-SUB)}\ \frac{\Gamma \vdash e : \tau^{r} \quad \Gamma \preceq \Gamma' \quad s \preceq r}{\Gamma' \vdash e : \tau^{s}} \qquad \text{(T-RET)}\ \frac{\Gamma \vdash v : \tau^{r}}{\Gamma \vdash \mathtt{return}\ v : \tau^{r}}\ r \neq \mathbf{0}$$

$$\text{(T-LET)}\ \frac{\Gamma_1 \vdash e_1 : \tau_1^{r_1} \quad \Gamma_2, x :_{r_1} \tau_1 \vdash e_2 : \tau_2^{r_2}}{\Gamma_1 + \Gamma_2 \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2^{r_2}}$$

$$\text{(T-APP)}\ \frac{\Gamma_1 \vdash v_1 : (\tau_1^{r_1} \to_{s} \tau_2^{r_2})^{(r + r \cdot s)} \quad \Gamma_2 \vdash v_2 : \tau_1^{r \cdot r_1}}{\Gamma_1 + \Gamma_2 \vdash v_1 v_2 : \tau_2^{r \cdot r_2}}\ r \neq \mathbf{0}$$

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

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

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

FIGURE 5.8 Graded type system

is satisfied by definition, whereas here we have to explicitly forbid to add a variable with a grade non-comparable with zero, such as, e.g., a linear variable.

In Figure 5.8, 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 Figure 5.8 define a unique judgment which is their union. We only comment the most significant points.

Rules (T-SUB-V) and (T-SUB) are as rule (T-SUB) in Figure 3.2, with the difference that here subtyping checks only the grades.

Rule (T-VAR) is analogous to that in Figure 3.2. Notably, 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.

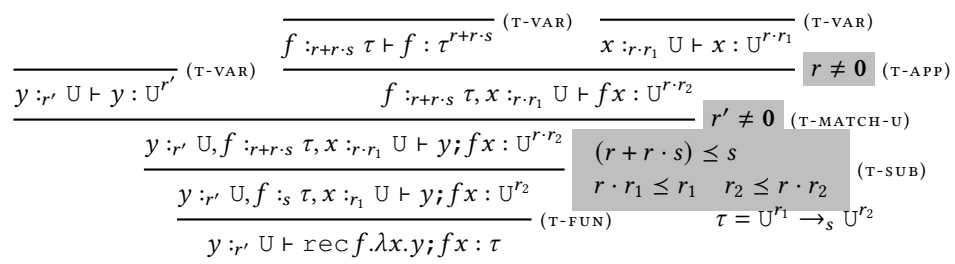

FIGURE 5.9 Example of type derivation: recursive function

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 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 5.2.1 below.

As in reduction rules in Figure 5.5, in the rules in the top section there is no constraint that the reduction grade should be non-zero; however, 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 5.2.1 : As an example of type derivation, we consider the function $\mathsf{div} = \mathtt{rec} f.\lambda x.y; fx$ introduced in Example 5.1.3. In Figure 5.9, 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 $(r + r \cdot s) \leq s$, 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.[5]. Assuming the grade algebra of natural numbers of Example 2.4.3(1), there is no grade s satisfying 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 5.1.3. On the other hand, for, e.g., $s = \infty$, with natural numbers extended as in Example 2.4.3(7), $(r + r \cdot s) \leq s$ 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 $r \cdot r_1 \leq r_1$ and $r_2 \leq r \cdot r_2$ would be always satified taking $r = 1$. Assuming the grade algebra of privacy

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

levels introduced in Example 3.2.3, where $\mathbf{1} = \mathsf{pub}$, for $s = \mathsf{pub}$ the condition is satisfied, again with no constraints on r_1 and r_2. For $s = \mathsf{priv}$, instead, it only holds for $r = \mathsf{priv}$, hence the condition $r_2 \preceq r \cdot r_2$ prevents the return type of the function to be pub, accordingly with the intuition that a function used in priv mode cannot return a pub 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 $\mathtt{U}^{r_2}$, 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 the 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 of the fact that the recursive function is 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, since the type system is a static overapproximation of the evaluation. We will show an example of this terminating resource-aware evaluation in Section 5.4 (Figure 5.15).

The following lemmas show that the promotion rule, usually explicitly stated in graded type systems, is admissible in our system (Lemma 5.2.6), and also a converse holds for value expressions (Lemma 5.2.7). We chose not to have an explicit promotion rule for the sake of simplicity in proofs, in fact, in this way we have one rule less. 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.[6] 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 5.2.2 (Inversion for value expressions):

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

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

6. If $\Gamma \vdash \texttt{inr}^{r_2} v : \tau^r$ then $r' \cdot \Gamma' \preceq \Gamma$, $\tau = \tau_1^{r_1} + \tau_2^{r_2}$ and $r \preceq r'$ such that $\Gamma' \vdash v : \tau_2^{r_2}$.

LEMMA 5.2.3 (Inversion for possibly diverging expressions):

1. If $\Gamma \vdash \texttt{return}\ v : \tau^r$ then $\Gamma' \vdash v : \tau^{r'}$ with $r \preceq r'$, $\Gamma' \preceq \Gamma$ 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 $\Gamma_1 + \Gamma_2 \preceq \Gamma$ and $r \preceq r'$.
3. If $\Gamma \vdash v_1 v_2 : \tau_2^r$ then $\Gamma_1 + \Gamma_2 \preceq \Gamma$ and $r \preceq r' \cdot r_2$ 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 $\Gamma_1 + \Gamma_2 \preceq \Gamma$ and $r \preceq t$ 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 $\Gamma_1 + \Gamma_2 \preceq \Gamma$ and $r \preceq t$ 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 $\Gamma_1 + \Gamma_2 \preceq \Gamma$ and $r \preceq s$ 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 $r' \neq \mathbf{0}$.

LEMMA 5.2.4 (Canonical Forms):

1. If $\Gamma \vdash v : (\tau_1^{r_1} \to s\tau_2^{r_2})^{r_3}$ then $v = \texttt{rec}\, f.\lambda x.e$.
2. If $\Gamma \vdash v : \texttt{Unit}$ then $v = \texttt{unit}$.
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 $v = \texttt{inl}^{r_1} v_1$ or $v = \texttt{inr}^{r_2} v_2$.

LEMMA 5.2.5 (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.

LEMMA 5.2.6 (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 $s \cdot \Gamma \vdash e : \tau^{s\cdot r}$.

Proof: By induction on the typing rules. We show only some cases.

(T-SUB-V) We have $\Delta \vdash v : \tau^{s'}$, $\Gamma \vdash v : \tau^r$, $r \preceq s'$ and $\Delta \preceq \Gamma$. By induction hypothesis $s \cdot \Delta \vdash v : \tau^{s\cdot s'}$. By monotonicity $s \cdot r \preceq s \cdot s'$ and $s \cdot \Delta \preceq s \cdot \Gamma$, 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 $\Gamma = r \cdot \Gamma'$, $v = \texttt{rec}\, f.\lambda x.e'$, $\tau = \tau_1^{r_1} \to s'\tau_2^{r_2}$ 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 $\Delta \vdash e : \tau^{s'}$, $\Gamma \vdash e : \tau^r$, $r \preceq s'$ and $\Delta \preceq \Gamma$. By induction

hypothesis $s \cdot \Delta \vdash e : \tau^{s \cdot s'}$. By monotonicity $s \cdot r \preceq s \cdot s'$ and $s \cdot \Delta \preceq s \cdot \Gamma$, 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.

□

LEMMA 5.2.7 (Demotion): If $\Gamma \vdash v : \tau^{s \cdot r}$ then $s \cdot \Gamma' \preceq \Gamma$ and $\Gamma' \vdash v : \tau^r$, for some Γ' .

Proof: By case analysis on v. We show only some cases.

$v = x$ By Lemma 5.2.2(1) $x :_{r'} \tau \preceq \Gamma$ and $s \cdot r \preceq r'$. Let $\Gamma' = x :_r \tau$. By monotonicity of $\cdot$ and, by transitivity of $\preceq$, $(s \cdot r) \cdot x :_1 \tau = s \cdot \Gamma' \preceq \Gamma$. By (T-VAR) $\Gamma' \vdash x : \tau^r$.

$v = \langle {}^{r_1} v_1, v_2 {}^{r_2} \rangle$ By Lemma 5.2.2(4) $r' \cdot (\Gamma_1 + \Gamma_2) \preceq \Gamma$ and $s \cdot r \preceq r'$ 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 $\preceq$ we have $s \cdot \Gamma' \preceq \Gamma$. By (T-PAIR) $\Gamma' \vdash v : \tau^r$.

□

LEMMA 5.2.8 (Substitution lemma): If $\Gamma, x :_r \tau' \vdash e : \tau^s$ and $\Delta \vdash v : \tau'^r$ then and $\Gamma + \Delta \vdash e[v/x] : \tau^r$

Proof: By induction on the structure of e. For the sake of brevity we will show only one case, other ones are analogous:

- $e = \texttt{let}\ y = e_1\ \texttt{in}\ e_2$
  By Lemma 5.2.3(2) we have $\Gamma_1, x :_{t_1} \tau' \vdash e_1 : \tau_1^{r_1}$ and $\Gamma_2, x :_{t_2} \tau', y :_{r_1} \tau_1 \vdash e_2 : \tau^{s'}$ and $\Gamma_1, x :_{t_1} \tau' + \Gamma_2, x :_{t_2} \tau' \preceq \Gamma, x :_r \tau'$ and $s \preceq s'$ and $t_1 + t_2 \preceq r$. By applying Lemma 5.2.7 and Lemma 5.2.6 we have $t_1 \cdot \Delta' \vdash v : \tau'^{t_1}$ and $t_2 \cdot \Delta' \vdash v : \tau'^{t_2}$ with $r \cdot \Delta' \preceq \Delta$. By induction hypothesis we have $\Gamma_1 + t_1 \cdot \Delta' \vdash e_1[v/x] : \tau_1^{r_1}$ and $\Gamma_2, y :_{r_1} \tau_1 + t_2 \cdot \Delta' \vdash e_2[v/x] : \tau^{s'}$. By applying (T-LET) we have $\Gamma_1 + (\Gamma_2, y :_{r_1} \tau_1) + (t_1 \cdot \Delta') + (t_2 \cdot \Delta') \vdash (\texttt{let}\ y = e_1\ \texttt{in}\ e_2)[v/x] : \tau^s$. We have $t_1 \cdot \Delta' + t_2 \Delta' = (t_1 + t_2 \cdot \Delta' \preceq r \cdot \Delta' \preceq \Delta$ so, by rule (T-SUB), we have $\Gamma_1 + (\Gamma_2, y :_{r_1} \tau_1) + \Delta \vdash (\texttt{let}\ y = e_1\ \texttt{in}\ e_2)[v/x] : \tau^s$, that is, the thesis.

□

In Figure 5.10 we give the typing rules for environments and configurations. In the rules, Γ is the context whose domain is the domain of the environment

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

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

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

FIGURE 5.10 Typing rules for environments and configurations

Each variable has as coeffect its grade in the environment, and as type the type of its value. The side conditions use the relation $\preceq_\bullet$ defined as follows:

$\Delta \preceq_\Theta \Gamma$ 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. We could alternatively typecheck values with grade r_i, as in Figure 3.5, and impose the condition $(\Gamma_1 + \ldots + \Gamma_n) \preceq_\bullet \Gamma$. Here we prefer this variant since more practical in the proofs.

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 2.4.3(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. As already said above, we plan to investigate how to impose a constraint also on the remaing resources in future work, see Chapter 8 for more details. This could be refined at the price of a more involved semantics.

5.3 Resource-aware 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 as described by Dagnino [23], Dagnino et al. [25], and De Nicola and Hennessy [27], 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 values and for 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 5.3.1 (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 5.3.2 (Progress/Subject reduction 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 $\hat{\Gamma}' + \Delta' \preceq_\Theta \Gamma'$.

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

Proof: By induction on the syntax of v.

- $v = x$ By Lemma 5.2.2(1) $x :_{r'} \tau \preceq \Delta$ with $r \preceq r'$. Since $\hat{\Gamma} + x :_{r'} \tau + \Theta \preceq \hat{\Gamma} + \Delta + \Theta \preceq \Gamma$ and, by (T-ENV), $\rho = x_1 \mapsto \langle \mathbf{v}_1, r_1 \rangle, \ldots, x_n \mapsto \langle \mathbf{v}_n, r_n \rangle$, $\Gamma = x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n = \Gamma', x :_s \tau, \Gamma_i \vdash \mathbf{v}_i : \tau_i^{r_i}\ \forall i \in 1..n$ and $t_1 + r' + t_2 \preceq s$ where $\mathtt{grade}(x, \hat{\Gamma}) = t_1$ and $\mathtt{grade}(x, \Theta) = t_2$. Let j be such that $x = x_j$, we get $\hat{\Gamma} = \hat{\Gamma}' + s \cdot \Gamma_j$ and $\Gamma_j \vdash \mathbf{v} : \tau^{\mathbf{1}}$. By Lemma 5.2.6 $r \cdot \Gamma_j \vdash \mathbf{v} : \tau^r$. By rule (VAR), $x|\rho', x \mapsto \langle \mathbf{v}, s \rangle \Rightarrow_r \mathbf{v}|\rho', x \mapsto \langle \mathbf{v}, t_1 + t_2 \rangle$. We have $\hat{\Gamma}' + (t_1 + t_2) \cdot \Gamma_j + r \cdot \Gamma_j + \Theta \preceq \hat{\Gamma}' + (t_1 + t_2) \cdot \Gamma_j + r' \cdot \Gamma_j + \Theta \preceq \hat{\Gamma} + \Theta$. We have $\mathtt{grade}(y, \hat{\Gamma} + \Theta) \preceq \mathtt{grade}(y, \Gamma')$ for all $y \in \mathrm{dom}(\hat{\Gamma} + \Theta) \setminus \{x\}$ and $\mathtt{grade}(x, \hat{\Gamma} + \Theta) = t_1 + t_2$, so $\hat{\Gamma} + \Theta \preceq \Gamma', x :_{t_1 + t_2} \tau$. 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 5.2.2(4) $r' \cdot (\Delta_1 + \Delta_2) \preceq \Delta$, $\tau = \tau_1^{r_1} \otimes \tau_2^{r_2}$ and $r \preceq r'$ such that $\Delta_1 \vdash v_1 : \tau_1^{r_1}$, $\Delta_2 \vdash v_2 : \tau_2^{r_2}$. By Lemma 5.2.6 $r \cdot \Delta_1 \vdash v_1 : \tau_1^{r \cdot r_1}$ and $r \cdot \Delta_2 \vdash v_2 : \tau_2^{r \cdot r_2}$. We have $\Gamma \vdash \rho \triangleright \hat{\Gamma}$ with $\hat{\Gamma} + r \cdot \Delta_1 + (r \cdot \Delta_2 + \Theta) \preceq \hat{\Gamma} + \Delta + \Theta \preceq \Gamma$. 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 $\hat{\Gamma}_1' + \Delta_1' + (r \cdot \Delta_2 + \Theta) \preceq \Gamma_1'$. Since $\Gamma_1' \vdash \rho_1' \triangleright \hat{\Gamma}_1'$ with $\hat{\Gamma}_1' + r \cdot \Delta_2 + (\Delta_1' + \Theta) \preceq \Gamma_1'$ 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 $\hat{\Gamma}_2' + \Delta_2' + (\Delta_1' + \Theta) \preceq \Gamma_2'$. By Lemma 5.2.7 $\Delta_1'' \vdash \mathbf{v}_1 : \tau_1^{r_1}$ and $\Delta_2'' \vdash \mathbf{v}_2 : \tau_2^{r_2}$ with $r \cdot \Delta_1'' \preceq \Delta_1'$ and $r \cdot \Delta_2'' \preceq \Delta_2'$. We have $\hat{\Gamma}_2' + r \cdot (\Delta_1'' + \Delta_2'') + \Theta = \hat{\Gamma}_2' + r \cdot \Delta_1'' + r \cdot \Delta_2'' + \Theta \preceq \hat{\Gamma}_2' + \Delta_2' + \Delta_1' + \Theta \preceq \Gamma_2'$. 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), $v|\rho \Rightarrow_r \langle {}^{r_1} \mathbf{v}_1, \mathbf{v}_2 {}^{r_2} \rangle | \rho_2'$.

□

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 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.

ADDING DIVERGENCE For possibly diverging expressions, instead, the big-step semantics defined in Figure 5.5 suffers from the long-known drawback shown by Cousot and Cousot [21] and Leroy and Grall [42] 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 Figure 5.5
- 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 Figure 5.11, 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 Figure 5.5 are emphasized in grey[7]).

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 what done by Ancona, Dagnino, and Zucca [3] and Dagnino [24]; here, for the reader's convenience, we provide a self-contained presentation, instantiating general definitions on our specific case.

Rules in Figure 5.11 handle divergence propagation. Notably, for each rule in the bottom section of Figure 5.5, 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 `let` x = e_1 `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 Figure 5.11, 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

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

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

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

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

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

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

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

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

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

$$(\textsc{co-div})\ \frac{}{e|\rho \Rightarrow_r \infty}$$

FIGURE 5.11 Adding divergence

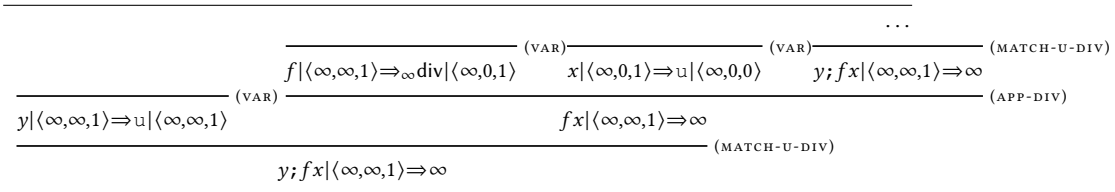

FIGURE 5.12 Example of resource-aware evaluation: divergency with no consumption

tree) from stuck computations (no proof tree). However, a purely coinductive interpretation would allow the derivation of spurious judgements, as shown by Ancona, Dagnino, and Zucca [3], Cousot and Cousot [21], and Leroy and Grall [42]. 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 5.3.4 below). Formally, we have the following definition instantiated from Ancona, Dagnino, and Zucca [3] and Dagnino [24].

DEFINITION 5.3.3 : A judgment $c \Rightarrow_r \mathrm{R}$ is derivable in the generalized inference system in Figure 5.11, 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 5.3.4 : Let us consider again the expression $y;fx$ of Example 5.1.3. Now, its non-terminating evaluation in the environment $y \mapsto \langle \mathtt{unit}, \infty \rangle, f \mapsto \langle \mathsf{div}, \infty \rangle, x \mapsto \langle \mathtt{unit}, 1 \rangle$, abbreviated $\langle \infty,\infty,1 \rangle$ using the previous convention, is formalized by the infinite proof tree in Figure 5.12, 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 $y;fx|\langle \infty,\infty,1 \rangle \Rightarrow \infty$ or $fx|\langle \infty,\infty,1 \rangle \Rightarrow \infty$, 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 $y;fx|\langle \infty,\infty,1 \rangle \Rightarrow \mathbf{v}|\langle \infty,\infty,1 \rangle$ 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 Figure 5.5 to that handling divergence in Figure 5.11 is an instance of a general construction, taking as input an arbitrary big-step semantics, fully formalized, and proved to be correct, by Dagnino [23]. In particular, the construction is *conservative*, that is, the semantics of converging computations is not affected [23].

DEFINITION 5.3.5 : Let $\vdash c \Rightarrow_r \mathbf{v}|\rho$ denote that the judgment can be derived by the rules in the bottom section of Figure 5.5, interpreted inductively.

THEOREM 5.3.6 (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 5.3.4, 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 Figure 5.11 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 5.3.7 (Soundness): If $\Gamma \vdash c : \tau^r$, then $\vdash_\infty c \Rightarrow_r \mathrm{R}$ 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 [57].

The proof is driven by coinductive reasoning on the semantic rules, following a schema firstly adopted by Ancona, Dagnino, and Zucca [3], 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 5.3.8 (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 $\langle r, c\rangle$ 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 [3, 24], 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 5.3.8 modularly relies on two results. The former (Theorem 5.3.9) is the instantiation of a general result proved (Theorem 3.3) by bounded coinduction [3]. 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 5.3.9 states completeness of diverging configurations with respect to any family of configurations which satisfies the *progress-∞ property*. The latter (Theorem 5.3.10) is the progress-∞ property for our type system.

THEOREM 5.3.9 (Progress-∞ $\Rightarrow$ 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:[8]

> (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 $c' \Rightarrow_s \mathrm{R}$, with $\mathrm{R} \neq \infty$, $\vdash c' \Rightarrow_s \mathrm{R}$.

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

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

1. S is consistent, that is, each $c \Rightarrow_r \mathrm{R}$ in S is the consequence of a rule whose premises are in S as well. We reason by cases:
   - 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 Figure 5.5 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 Figure 5.11. 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 5.3.8) 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 5.3.9, $WT_r^\infty = \{c \mid c \in WT^r \text{ and } \nexists\, \mathbf{v}|\rho \text{ such that } \vdash_\infty c \Rightarrow_r \mathbf{v}|\rho\}$.

THEOREM 5.3.10 (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 $\mathrm{R} \neq \infty$, $\vdash c' \Rightarrow_s \mathrm{R}$.

8 Keep in mind that $c \Rightarrow_r \mathrm{R}$ denotes just the judgment (a triple), whereas $\vdash c \Rightarrow_r \mathrm{R}$ and $\vdash_\infty c \Rightarrow_r \mathrm{R}$ denote derivability of the judgment (Definition 5.3.5 and Definition 5.3.3, respectively).

Proof: By case analysis on c. We show only one case for the sake of brevity, since other ones are similar.

$c = \texttt{let}\ x = e_1\ \texttt{in}\ e_2|\rho$

By rule (T-CONF) we have $\Delta \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau^r$ and $\Gamma \vdash \rho \triangleright \hat{\Gamma}$ with $\hat{\Gamma} + \Delta \preceq_\Theta \Gamma$ for a given Θ. By Lemma 5.2.3(2) we have $\Delta_1 \vdash e_1 : \tau_1^{r_1}$ 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 two cases:

- $\nexists\ \mathbf{v}_1|\rho_1$ such that $e_1|\rho \Rightarrow_{r_1} \mathbf{v}_1|\rho_1$
  Since $\hat{\Gamma} + \Delta_1 \preceq_{\Theta+\Delta_2} \Gamma$, by applying rule (LET-DIV1) we have the thesis.
- $\exists\ \mathbf{v}_1|\rho_1$ such that $e_1|\rho \Rightarrow_{r_1} \mathbf{v}_1|\rho_1$
  Since we have $e_1|\rho \Rightarrow_{r_1} \mathbf{v}_1|\rho_1$ we also have $e_1|\tilde{\rho} \Rightarrow \mathbf{v}_1|\tilde{\rho}_1$ with $\lceil\rho\rceil = \tilde{\rho}$ and $\lceil\rho_1\rceil = \tilde{\rho}_1$. We have $\hat{\Gamma} + \Delta_1 \preceq_{\Theta+\Delta_2} \Gamma$. By Lemma 5.3.11 we have $\Delta_1' \vdash \mathbf{v}_1 : \tau_1^r$ and $\Gamma_1 \vdash \rho_2 \triangleright \hat{\Gamma}_1$ with $\hat{\Gamma}_1 + \Delta_1' \preceq_{\Theta+\Delta_2} \Gamma_1$, for some ρ_2 such that $\lceil\rho_2\rceil = \tilde{\rho}_2$ and $e_1|\rho_1 \Rightarrow_{r_1} \mathbf{v}_1|\rho_2$. By Lemma 5.2.7 we have $\Delta_1'' \vdash \mathbf{v} : \tau^1$ with $r_1 \cdot \Delta_1'' \preceq \Delta_1'$. We have $\hat{\Gamma}_1 + r_1 \cdot \Delta_1'' \preceq_{\Theta+\Delta_2} \hat{\Gamma}_1 + \Delta_1' \preceq_{\Theta+\Delta_2} \Gamma_1$. Since x' is fresh we have $\hat{\Gamma}_1 + r_1 \cdot \Delta_1'' \preceq_{\Theta+\Delta_2+x':_{r_1}\tau_1} \Gamma_1 + x' :_{r_1} \tau_1$, we have $\Gamma, x' :_{r_1} \tau_1 \vdash \rho_2, x' \mapsto \langle r_1, \mathbf{v}\rangle \triangleright \hat{\Gamma} + r_1 \cdot \Delta_1''$. By Lemma 5.2.8 and by knowing that $x' :_{r_1} \tau_1 \vdash x' : \tau_1^{r_1}$ we have $\Delta_2, x' :_{r_1} \tau_1 \vdash e_2[x'/x] : \tau^{r'}$. Since we have $\hat{\Gamma}_1 + r_1 \cdot \Delta_1'' + \Delta_2, x' :_{r_1} \tau_1 \preceq_\Theta \Gamma_1, x' :_{r_1} \tau_1$ by rule (T-CONF) we have $\Gamma_1, x' :_{r_1} \tau_1 \vdash e_2[x'/x]|\rho_2, x' \mapsto \langle r_1, \mathbf{v}\rangle : \tau^{r'}$. If $\nexists\ \mathbf{v}_2|\rho_3$ such that $e_2[x'/x]|\rho_2, x' \mapsto \langle r_1, \mathbf{v}\rangle \Rightarrow_{r'} \mathbf{v}_2|\rho_3$ then by rule (LET/LET-DIV2) we have the thesis, otherwise we have an absurd, since if $\mathbf{v}_2|\rho_3$ would exists then, by rule (LET/LET-DIV2) also $c \Rightarrow_r \mathbf{v}_2|\rho_3$.

□

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 $\lceil\rho\rceil$ 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 Figure 5.5 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*[9] decorated with grades to be well-typed, with the same residual context and we can reduce the initial configuration to the well typed environment with graded reduction. Note that, differently from Lemma 5.3.2, here the hypothesis of well-typedness of the configuration is not enough, but we need also to assume progress of standard reduction. a

9 That is, as soundness, subject reduction holds in the *may* flavour.

LEMMA 5.3.11 (Subject reduction): If $\Delta \vdash e : \tau^r$ and $\Gamma \vdash \rho_1 \triangleright \hat{\Gamma}$ with $\hat{\Gamma} + \Delta \preceq_\Theta \Gamma$ and, set $\tilde{\rho}_1 = \lceil \rho_1 \rceil$, $e|\tilde{\rho}_1 \Rightarrow \mathbf{v}|\tilde{\rho}_2$, 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 $\lceil \rho_2 \rceil = \tilde{\rho}_2$ and $e|\rho_1 \Rightarrow_r \mathbf{v}|\rho_2$.

Proof: By induction on the reduction $e|\tilde{\rho}_1 \Rightarrow v|\tilde{\rho}_2$.

(RET) By Lemma 5.2.3(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 5.3.2 $v|\rho_1 \Rightarrow_r \mathbf{v}|\rho_2'$ 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) $\texttt{return}\ v|\rho_1 \Rightarrow_r \mathbf{v}|\rho_2'$. By $\texttt{return}\ v|\rho_1 \Rightarrow_r \mathbf{v}|\rho_2'$, we derive $\texttt{return}\ v|\tilde{\rho}_1 \Rightarrow \mathbf{v}|\tilde{\rho}_2'$. Since $\Rightarrow$ is deterministic, $\tilde{\rho}_2' = \tilde{\rho}_2$, that is, the thesis.

(LET) By Lemma 5.2.3(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 $\lceil \rho_2 \rceil = \tilde{\rho}_2$ and $e_1|\rho_1 \Rightarrow_{r_1} \mathbf{v}|\rho_2$. By this relation we derive $\hat{\Gamma}' + \Delta_2 \preceq_{\Theta+\Delta'} \Gamma'$. By Lemma 5.2.7 $r_1 \cdot \Delta'' \preceq \Delta'$ and $\Delta'' \vdash \mathbf{v} : \tau_1^1$. 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 $\Gamma', x' :_{r_1} \tau_1 \vdash \rho_2, x' \mapsto \langle \mathbf{v}, r_1 \rangle \triangleright \hat{\Gamma}' + r_1 \cdot \Delta''$. 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 $\Delta_2, x :_{r_1} \tau_1 \vdash e_2 : \tau^{r'}$ and by applying (T-SUB) 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 $\lceil \rho_3 \rceil = \tilde{\rho}_3$ and $e_2[x'/x]|\tilde{\rho}_2, x' \mapsto \langle r_1, \mathbf{v} \rangle \Rightarrow_r \mathbf{v}'|\rho_3$. Since $e_1|\rho_1 \Rightarrow_{r_1} \mathbf{v}|\rho_2$ and $e_2[x'/x]|\tilde{\rho}_2, x' \mapsto \langle r_1, \mathbf{v} \rangle \Rightarrow_r \mathbf{v}'|\rho_3$ by (LET) we have $\texttt{let}\ x = e_1\ \texttt{in}\ e_2|\rho_1 \Rightarrow_r \mathbf{v}|\rho_3$ and so $\texttt{let}\ x = e_1\ \texttt{in}\ e_2|\tilde{\rho}_1 \Rightarrow \mathbf{v}|\tilde{\rho}_3'$. Since $\Rightarrow$ is deterministic, $\tilde{\rho}_3 = \rho_3'$ and we have the thesis.

(APP) We have $v_1|\tilde{\rho}_1 \Rightarrow \texttt{rec}\, f.\lambda x.e|\tilde{\rho}_2$, $v_2|\tilde{\rho}_2 \Rightarrow \mathbf{v}_2|\tilde{\rho}_3$ and $e[x'/x][f'/f]|\tilde{\rho}_3, x' : \mathbf{v}_2, f' : \texttt{rec}\, f.\lambda x.e \Rightarrow \mathbf{v}|\tilde{\rho}_4$ with x', f' fresh. By Lemma 5.2.3(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 5.3.2 and by Lemma 5.2.4(1) $v_1|\rho_1 \Rightarrow_{r'+r'\cdot s} \texttt{rec}\, f.\lambda x.e|\rho_2$ and $\Delta_1' \vdash \texttt{rec}\, f.\lambda x.e : (\tau_1^{r_1} \to_s \tau_2^{r_2})^{r'+r'\cdot s}$ 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 $\lceil \rho_2 \rceil = \tilde{\rho}_2$. We also have $\hat{\Gamma}' + \Delta_2 \preceq_{\Theta+\Delta_1'} \Gamma'$, so, by Lemma 5.3.2 $v_2|\rho_2 \Rightarrow_{r'\cdot r_1} \mathbf{v}_2|\rho_3$ 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 $\lceil \rho_3 \rceil = \tilde{\rho}_3$. By Lemma 5.2.2(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) $\Delta_1'' \vdash \texttt{rec}\, f.\lambda x.e : \tau_1^1$. By Lemma 5.2.8 and by Lemma 5.2.6 $r' \cdot (\Delta_1'', f' :_s \tau_1^{r_1} \to_s \tau_2^{r_2}, x' :_{r_1} \tau_1) \vdash e[f'/f][x'/x] : \tau_2^{r'\cdot r_2}$. By Lemma 5.2.7 $\Phi_2 \vdash \mathbf{v}_2 : \tau_2^1$ 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) $\Gamma'', x' :_{r'\cdot r_1} \tau_1, f' :_{r'\cdot s} (\tau_1^{r_1} \to_s \tau_2^{r_2}) \vdash \rho_3, x' \mapsto \langle \mathbf{v}_2, r' \cdot r_1\rangle, f' \mapsto \langle \mathtt{rec}\, f.\lambda x.e, r' \cdot s\rangle \triangleright \hat{\Gamma}'' + r'\cdot s\cdot\Phi_1 + r'\cdot r_1\cdot\Phi_2 + r'\cdot(\Delta_1'', f' :_s \tau_1^{r_1} \to_s \tau_2^{r_2}, x' :_{r_1} \tau_1)$. 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 have $\Delta' \vdash \mathbf{v} : \tau^{r\cdot r_2}$ and $\Gamma' \vdash \rho_4 \triangleright \hat{\Gamma}'$ with $\hat{\Gamma}' + \Delta' \preceq_\Theta \Gamma'$, for some $\tilde{\rho}_4$ such that $\lceil \rho_4 \rceil = \tilde{\rho}_4$ and $e[x'/x][f'/f]|\tilde{\rho}_3, x' : \mathbf{v}_2, f' : \mathtt{rec}\, f.\lambda x.e \Rightarrow_{r'\cdot r_2} \mathbf{v}|\rho_4$. By applying rule (APP) we have $v_1 v_2|\rho_1 \Rightarrow_r \mathbf{v}|\rho_4$ and so also $v_1 v_2|\tilde{\rho}_1 \Rightarrow \mathbf{v}|\tilde{\rho}_4'$. Since $\Rightarrow$ is deterministic $\tilde{\rho}_4 = \tilde{\rho}_4'$ we have the thesis.

(MATCH-P) By Lemma 5.2.3(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 5.3.2 $v|\rho \Rightarrow_s \mathbf{v}|\rho'$ 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 5.2.4(3) $\mathbf{v} = \langle {}^{r_1}\mathbf{v}_1, \mathbf{v}_2\, {}^{r_2}\rangle$. By Lemma 5.2.2(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 5.2.7 $\Phi_1 \vdash \mathbf{v}_1 : \tau_1^{\mathbf{1}}$ 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 5.2.8 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) $\Gamma', x :_{s\cdot r_1} \tau_1, y :_{s\cdot r_2} \tau_2 \vdash \rho_1, x' \mapsto \langle \mathbf{v}_1, s \cdot r_1\rangle, y' \mapsto \langle \mathbf{v}_2, s \cdot r_2\rangle \triangleright \hat{\Gamma}' + (s \cdot r_1) \cdot \Phi_1 + (s \cdot r_2) \cdot \Phi_2$. By induction hypothesis on $e[x'/x][y'/y]|\rho_1, x' \mapsto \langle \mathbf{v}_1, s \cdot r_1\rangle, y' \mapsto \langle \mathbf{v}_2, s \cdot r_2\rangle \Rightarrow \mathbf{v}'|\rho_2$ we get $\Delta' \vdash \mathbf{v}' : \tau^r$ and $\Gamma' \vdash \rho_2 \triangleright \hat{\Gamma}'$ with $\hat{\Gamma}' + \Delta' \preceq_\Theta \Gamma'$, for some ρ_2 such that $\lceil \rho_2 \rceil = \tilde{\rho}_2$ and $e[x'/x][y'/y]|\rho_1, x' \mapsto \langle \mathbf{v}_1, s \cdot r_1\rangle, y' \mapsto \langle \mathbf{v}_2, s \cdot r_2\rangle \Rightarrow_r \mathbf{v}|\rho_2$. By applying rule (MATCH-P) and by determinism of $\Rightarrow$ we have the thesis.

□

5.4 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 $\mathtt{match}\ e\ \mathtt{with}\ \ell_1 x_1\ \mathtt{or}\ldots\ \mathtt{or}\ \ell_n x_n$. We write just ℓ as an abbreviation for an addend $\ell{:}\mathtt{Unit}^{\mathbf{0}}$ in a variant type, and also for the corresponding tagged variants $\ell^{\mathbf{0}}\mathtt{unit}$ and patterns ℓx 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 $\mathbf{1}$ annotations as in the previous examples, and we will consider $\mathbf{0}$ as default, hence omitted, as recursion grade (that is, functions are by default non-recursive).

```
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
```

FIGURE 5.13 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 Figure 5.13 followed by the definition of some standard functions. Assume, firstly, the grade algebra of natural numbers with bounded usage, Example 2.4.3(1), extended with ∞, as in Example 2.4.3(7), needed as annotation of recursive functions, as has been illustrated in Example 5.2.1; 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.

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^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 2.4.3(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 Figure 5.7 is the *tensor* product, also called *multiplicative*, following Linear Logic terminology of Girard [34]. 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[10] rule (MATCH-P) in Figure 5.5. 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 Figure 5.5. The *cartesian* product, instead, also called *additive*, formalized in Figure 5.14, has one destructor for each component, so that the component 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

10 Typing rules follow the same pattern.

$$\begin{array}{lcl} \mathbf{v} & ::= & \ldots \mid [^{r_1}\mathbf{v}_1, \mathbf{v}_2{}^{r_2}] \mid \pi_1\mathbf{v} \mid \pi_2\mathbf{v} \\ \sigma, \tau & ::= & \ldots \mid T \times S \end{array}$$

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

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

FIGURE 5.14 Cartesian product

$$\dfrac{\dfrac{}{\texttt{ev}\,|\langle 1,1,\texttt{z}\rangle \Rightarrow_1 \texttt{even}|\langle 0,1,\texttt{z}\rangle}\,\text{(VAR)} \quad \dfrac{}{\texttt{n}|\langle 0,1,\texttt{z}\rangle \Rightarrow \texttt{z}|\langle 0,0,\texttt{z}\rangle}\,\text{(VAR)} \quad \mathcal{D}}{\texttt{ev n}|\langle 1,1,\texttt{z}\rangle \Rightarrow \texttt{t}|\langle 0,0,\texttt{z}\rangle}\,\text{(APP)}$$

$$\dfrac{\dfrac{}{\texttt{ev}\,|\langle n,1,\texttt{s}^n\texttt{z}\rangle \Rightarrow_n \texttt{even}|\langle n-1,1,\texttt{z}\rangle}\,\text{(VAR)} \quad \dfrac{}{\texttt{n}|\langle n-1,1,\texttt{z}\rangle \Rightarrow \texttt{z}|\langle n-1,0,\texttt{z}\rangle}\,\text{(VAR)} \quad \mathcal{D}'}{\texttt{ev n}|\langle n,1,\texttt{s}^n\texttt{z}\rangle \Rightarrow b|\langle 0,0,\texttt{z}\rangle}\,\text{(APP)}$$

$$\text{with } \mathcal{D} = \dfrac{\texttt{t}|\langle 0,0,\texttt{z}\rangle \Rightarrow \texttt{t}|\langle 0,0,\texttt{z}\rangle}{\texttt{if-z (z, t, sn}\rightarrow\texttt{not(ev n))}|\langle 0,0,\texttt{z}\rangle \Rightarrow \texttt{t}|\langle 0,0,\texttt{z}\rangle}\,\text{(MATCH-L)}$$

$$\text{and } \mathcal{D}' = \dfrac{\texttt{not(ev n)}|\langle n-1,1,\texttt{z}\rangle \Rightarrow \overline{b}|\langle 0,0,\texttt{z}\rangle}{\texttt{if-z (z, t, sn}\rightarrow\texttt{not(ev n))}|\langle n-1,1,\texttt{z}\rangle \Rightarrow b|\langle 0,0,\texttt{z}\rangle}\,\text{(MATCH-L)}$$

FIGURE 5.15 Example of resource-aware evaluation: terminating recursion

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 Figure 5.15.

For brevity, we write `z`, `s`, and `t` for `zero`, `succ`, and `true`, respectively, `if-z (`$v, e_1,$ `s`$x\rightarrow e_2$`)` for `match` v `with z` $\rightarrow e_1$ `or s`$x \rightarrow e_2$, and $\langle r,s,\mathbf{v}\rangle$ for the environment `ev` $\mapsto \langle$`even`$, r\rangle$, `n` $\mapsto \langle \mathbf{v}, s\rangle$. 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 $\rightarrow$ 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 $\texttt{s}^n\texttt{z}$, for $n \neq 0$. Here, b stands for either `true` or `false`, and $\overline{b}$ for its complement.

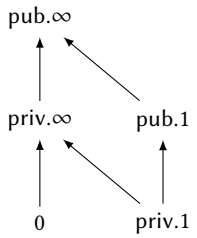

FIGURE 5.16 Grade algebra of privacy levels and linearity

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

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

FIGURE 5.17 Types of data and filesystem interface

PROCESSING DATA FROM/TO FILES The following example illustrates how we can simultaneously track privacy levels, as introduced in Example 5.1.2, 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 2.4.7. So there are five grades: 0 (meaning unused), priv.1 and pub.1 (meaning used linearly in either priv or pub mode), and priv.∞, pub.∞ (meaning used an arbitrary number of times in either priv or pub mode). The partial order is graphically shown in Figure 5.16. The neutral element for multiplication is pub.1, which therefore will be omitted.

In Figure 5.17 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 Figure 5.18 we use, rather than `match` e_1 `with unit` $\rightarrow e_2$, the alternative syntax e_1 ; e_2 mentioned in Section 5.1. 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.∞ FileHandle → fnTypepub.∞ → Resultpub.∞=
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
```

FIGURE 5.18 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 Figure 5.16, 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.∞⊗ 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.∞ → (ok:Charpub.∞ + 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.∞ FileHandle → fnTypepub.∞ → Resultpriv.∞
```

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

6

Beyond structural coeffects

In the type systems presented in previous chapters, coeffect contexts are (representations of) finite maps from variables to grades, called coeffects when used in this position. As firstly noted by McBride [46] and Wood and Atkey [56], and formalized at the end of Section 2.4, such coeffect contexts form a (partially ordered) *module* over the underlying grade algebra, that is, they are equipped with partial order, sum, zero, and scalar multiplication, satisfying the required axioms, which are defined pointwise on top of the corresponding ones of the grade algebra. Intuitively, this means that the coeffects of each variable (the way it is used) can be computed independently; when this happens, coeffects are called *structural*.

The aim of this chapter is to investigate a significant example in which structural coeffects are not adequate. Notably, we want to use coeffects to statically guarantee relevant properties on the usage of memory in an imperative language. It is importante to notice that the same approach could be used also in OO and functional languages.

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

A likely reason is that this kind of coeffect does not fit in the framework of structural coeffect contexts. The problem is that sharing is propagated transitively: for instance, a program introducing sharing between x and y, and between y and z, introduces sharing between x and z as well.

In this chapter, we show that sharing can be naturally modeled by adopting coeffect contexts which are *non-structural*, but still have a module structure, notably providing sum and scalar multiplication. The idea is that operations on coeffect contexts are not simply the pointwise extension of grade operations, since sharing information should transitively propagated. In other words, the coeffect is not a property of a single variable, but of the whole context, as in *flat* coeffects [52], which, however, have not even a per-variable representation. As a consequence, coeffects cannot be the same of grades used as annotations of types, as we did in the previous chapters; hence, in this chapter we will propose a type-and-coeffect system rather than a graded type system, leaving to future work the integration with graded types.

In such type-and-coeffect system, we are able to detect, in a simple and

static way, some relevant notions in the literature, notably that an expression is a *capsule*, that is, evaluates to the *unique entry point* for a portion of memory. To illustrate its effectiveness, we enhance the type system tracking sharing to model a sophisticated set of features related to uniqueness and immutability [32, 33].

We illustrate the approach on a simple reference language, an imperative variant of the calculus considered in Chapter 3.

In Section 6.1 we briefly describe the language, and in Section 6.2 we introduce the notions about sharing and immutability needed to express the properties we want to enforce. In Section 6.3 and Section 6.4 we describe the two type systems with the related results. Finally, in Section 6.5 we discuss the expressive power, comparing with closely related proposals.

The type system presented in this chapter overapproximate sharing, as in may-alias analysis, but a deep inverstigation to understand how to manage must-alias analysis is a possible future work.

6.1 Imperative Java-like calculus

Syntax, reduction rules, and the standard type system are reported in Figure 6.1.

The syntax is an extension of that in Figure 3.1. Notably, here we have primitive types, included to show that they that behave as immutable references, and, since we are in an imperative context, field assignment.

Expressions of primitive types, unspecified, include constants k. Variables $x, y, z, \ldots$ occur both in source code (method parameters, including the special variable `this`, and local variables in blocks) and as references in memory. We will sometimes abbreviate $\{\tau\ x = e;\ e'\}$ by $e; e'$ when x does not occur free in e.

The class table is abstractly modeled by the functions fields, mbody, and mtype, defined as in Section 3.1. For simplicity, here we do not consider subtyping (inheritance), which is an orthogonal feature.

Reduction is defined over *configurations* of shape $e|\mu$, where a *memory* μ is a map from references to *objects* of shape $[v_1, \ldots, v_n]^C$, and we assume free variables in e to be in $\mathsf{dom}(\mu)$. We denote by $\mu^{x.i=v}$ the memory obtained from μ by updating the i-th field of the object associated to x by v. Note that, being the calculus imperative, values of class types are now references in memory.

Reduction and typing rules are straightforward. Differently from Chapter 3, here for simplicity we use evaluation contexts in reduction rules. In rule (T-MEM), a memory is well-formed in a context assigning a type to all and only references in memory, provided that, for each reference, the associated object has the same type. In rule (T-CONF), a configuration is well-typed if the expression is well-typed, and the memory is well-formed in the same context (recall that free variables in the expression are bound in the domain of the memory).

$$
\begin{array}{lcll}
e & ::= & x \mid k \mid e.f \mid e.f = e' \mid \texttt{new}\ C(es) \mid & \text{expression} \\
 & \mid & e.m(es) \mid \{\tau\ x = e;\ e'\} \mid \ldots & \\
\tau & ::= & C \mid P & \text{type} \\
 & & & \\
v & ::= & x \mid k & \text{value} \\
\mathcal{E} & ::= & [\,] \mid \mathcal{E}.f \mid \mathcal{E}.f = e' \mid x.f = \mathcal{E} \mid \texttt{new}\ C(vs, \mathcal{E}, es) & \text{evaluation context} \\
 & \mid & \mid \mathcal{E}.m(es) \mid x.m(vs, \mathcal{E}, es) \mid \{\tau\ x = \mathcal{E};\ e\} \mid \ldots &
\end{array}
$$

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

$$
\text{(FIELD-ACCESS)}\ \frac{}{x.f_i|\mu \to v_i|\mu}\ \begin{array}{l}\mu(x) = \texttt{new}\ C(v_1, \ldots, v_n) \\ \mathsf{fields}(C) = \tau_1\ f_1;\ \ldots \tau_n\ f_n; \\ i \in 1..n\end{array}
$$

$$
\text{(FIELD-ASSIGN)}\ \frac{}{x.f_i = v|\mu \to v|\mu^{x.i=v}}\ \begin{array}{l}\mu(x) = [v_1, \ldots, v_n]^C \\ \mathsf{fields}(C) = \tau_1\ f_1;\ \ldots \tau_n\ f_n; \\ i \in 1..n\end{array}
$$

$$
\text{(NEW)}\ \frac{}{\texttt{new}\ C(vs)|\mu \to x|\mu, x \mapsto \texttt{new}\ C(vs)}\ x \notin \mathsf{dom}(\mu)
$$

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

$$
\text{(BLOCK)}\ \frac{}{\{\tau\ x = v;\ e\}|\mu \to e[v/x]|\mu}
$$

$$
\text{(T-VAR)}\ \frac{}{\Gamma \vdash x : \tau}\ \Gamma(x) = \tau \qquad \text{(T-CONST)}\ \frac{}{\Gamma \vdash k : P_k}
$$

$$
\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash e : C}{\Gamma \vdash e.f_i : \tau_i}\ \begin{array}{l}\mathsf{fields}(C) = \tau_1\ f_1;\ \ldots \tau_n\ f_n; \\ i \in 1..n\end{array}
$$

$$
\text{(T-FIELD-ASSIGN)}\ \frac{\Gamma \vdash e : C \quad \Gamma \vdash e' : \tau_i}{\Gamma \vdash e.f_i = e' : \tau_i}\ \begin{array}{l}\mathsf{fields}(C) = \tau_1\ f_1;\ \ldots \tau_n\ f_n; \\ i \in 1..n\end{array}
$$

$$
\text{(T-NEW)}\ \frac{\Gamma \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Gamma \vdash \texttt{new}\ C(e_1, \ldots, e_n) : C}\ \mathsf{fields}(C) = \tau_1\ f_1;\ \ldots \tau_n\ f_n;
$$

$$
\text{(T-INVK)}\ \frac{\Gamma \vdash e_0 : C \qquad \Gamma \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Gamma \vdash e_0.m(e_1, \ldots, e_n) : \tau}\ \mathsf{mtype}(C, m) = \tau_1 \ldots \tau_n \to \tau
$$

$$
\text{(T-BLOCK)}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \{\tau\ x = e;\ e'\} : \tau'}
$$

$$
\text{(T-OBJ)}\ \frac{\Gamma \vdash v_i : \tau_i \quad \forall i \in 1..n}{\Gamma \vdash [v_1, \ldots, v_n]^C : C}\ \mathsf{fields}(C) = \tau_1\ f_1;\ \ldots \tau_n\ f_n;
$$

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

$$
\text{(T-CONF)}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \mu}{\Gamma \vdash e|\mu : \tau}
$$

FIGURE 6.1 Syntax, standard reduction, and standard type system

6.2 Sharing and mutation

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

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

DEFINITION 6.2.1 (Sharing in memory): The *sharing relation* in memory μ, denoted by $\bowtie_\mu$, is the smallest equivalence relation on $\mathrm{dom}(\mu)$ such that:

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

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

It is important for a programmer to be able to rely on *capsule* and *immutability* properties. Informally, an expression has the capsule property if its result will be the *unique entry point* for a portion of memory. For instance, we expect the result of a `clone` method to be a capsule, see Example 6.2.5 below. This allows programmers to identify state that can be safely used by a thread since no other thread can access/modify it. A reference has the immutability property if its reachable object graph will be never modified. As a consequence, an immutable reference can be safely shared by threads.

The following simple example illustrates the capsule property.

EXAMPLE 6.2.2 : Assume the following class table:

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

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

Lent and capsule properties are formally defined below. We denote by $\mathrm{fv}(e)$ the free variables of e.

DEFINITION 6.2.3 (Lent reference): For $x \in \mathrm{fv}(e)$, x is lent in e if, for all μ, $e|\mu \rightarrow^\star y|\mu'$ implies $x \bowtie_{\mu'} y$ does not hold.

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

DEFINITION 6.2.4 (Capsule expression): An expression e is a *capsule* if, for all μ, $e|\mu \rightarrow^\star y|\mu'$ implies that, for all $x \in \mathrm{fv}(e)$, $x \bowtie_{\mu'} y$ does not hold.

The capsule property can be easily detected in simple situations, such as using a primitive deep clone operator, or a closed expression. However, the property also holds in many other cases, which are *not* easily detected (statically) since they depend on *the way variables are used*. To see this, we consider a more involved example, adapted from what done by Giannini et al. [33].

EXAMPLE 6.2.5 :

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

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

Now let us consider the code after the class definition, where the programmer wants the guarantee that `mycaps` will be initialized with a capsule, that is, an expression which evaluates to the entry point of a fresh portion of memory. Figure 6.2 shows a graphical representation of the memory after

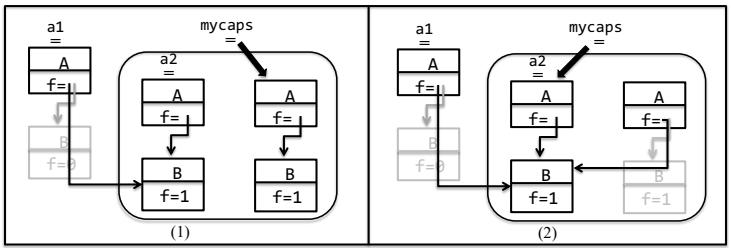

FIGURE 6.2 Graphical representation of the memory for Example 6.2.5

the evaluation of such code. Side (1) shows the resulting memory if we evaluate line (1) but not line (2), while side (2) shows the resulting memory if

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

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

DEFINITION 6.2.6 : The *reachability relation* in memory μ, denoted by $\triangleright_\mu$, is the reflexive and transitive closure of the relation on $\mathrm{dom}(\mu)$ such that:

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

DEFINITION 6.2.7 (Immutable reference): For $x \in \mathrm{fv}(e)$, x is *immutable* in e if $e|\mu \rightarrow^\star e'|\mu'$ and $x \triangleright_\mu y$ implies $\mu(y) = \mu'(y)$.

A typical way to prevent mutation, as we will show in Section 6.4, is by a type modifier read, so that an expression with type tagged in this way cannot occur as the left-hand side of a field assignment. However, to have the guarantee that a certain portion of memory is actually immutable, a type system should be able to detect that it cannot be modified through *any* possibile reference. For instance, consider a variant of Example 6.2.5 with the same classes `A` and `B`.

EXAMPLE 6.2.8 :

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

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

6.3 Coeffects for sharing

Introducing sharing, e.g., by a field assignment $x.f = y$, can be clearly seen as adding an arc between x and y in an undirected graph where nodes are variables. However, such a graphical representation would be a *global* one, whereas we would like to keep a *per variable* representation, which, moreover, supports sum and scalar multiplication operators. We introduce auxiliary entities called *links*, and attach to each variable a set of them, so that an arc between x and y is represented by the fact that they have a common link[1]. Moreover, there is a special link R which denotes a connection with the final result of the expression.

For instance, considering again the classes of Example 6.2.2:

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

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

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

meaning that the program's execution introduces sharing between `x` and `y`, as expressed by their common link ℓ, and between `z1`, `z2`, and the final result, as expressed by their common link R. The derivation for this judgment is shown later (Figure 6.4).

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

$$\ell \in X_i \text{ implies } \ell \in X_i^\star$$
$$\ell, \ell' \in X_i^\star, \ell' \in X_j^\star \text{ implies } \ell \in X_j^\star$$

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

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

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

Scalar multiplication is defined in terms of an operator $\triangleleft$ on sharing coeffects, which replaces the R link (if any) in the second argument with the first argument:

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

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

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

Altogether we have the following formal definition:

DEFINITION 6.3.1 : The *sharing coeffect system* is defined by:

- the grade algebra $\mathcal{L} = \langle L, \subseteq, \cup, \triangleleft, \emptyset, \{\mathrm{R}\}\rangle$
- the partially ordered $\mathcal{L}$-module $\langle \mathrm{CCtx}^L_\star, \hat{\subseteq}, +, \emptyset, \cdot\rangle$ where:
  - $\mathrm{CCtx}^L_\star$ are the fixpoints of $\star$, that is, the closed coeffect contexts
  - $\hat{\subseteq}$ is the pointwise extension of $\subseteq$ to $\mathrm{CCtx}^L_\star$
  - $\gamma + \gamma' = (\gamma \,\hat{\cup}\, \gamma')^\star$, where $\hat{\cup}$ is the pointwise extension of $\cup$ to $\mathrm{CCtx}^L_\star$
  - $X \cdot \gamma = (X \,\hat{\triangleleft}\, \gamma)^\star$, where $\hat{\triangleleft}$ is the pointwise extension of $\triangleleft$ to $\mathrm{CCtx}^L_\star$.

Operations on closed coeffect contexts can be lifted to type-and-coeffect contexts in the usual way.

It is easy to check that $\mathcal{L} = \langle L, \subseteq, \cup, \triangleleft, \emptyset, \{\mathrm{R}\}\rangle$ is actually a grade algebra with $\emptyset$ neutral element of $\cup$ and $\{\mathrm{R}\}$ neutral element of $\triangleleft$. The fact that $\langle \mathrm{CCtx}^L_\star, \hat{\subseteq}, +, \emptyset, \cdot\rangle$ is actually a partially ordered $\mathcal{L}$-module can be proved as follows: first of all, $\mathcal{L}^V = \langle \mathrm{CCtx}^L, \hat{\subseteq}, \hat{\cup}, \emptyset, \hat{\triangleleft}\rangle$ is a partially ordered $\mathcal{L}$-module, notably, the structural one (all operations are pointwise); it is easy to see that $\_^\star$ is an idempotent homomorphism on $\mathcal{L}^V$; then, the thesis follows from a result proved by Bianchini et al. [12], stating that an idempotent homomorphism on a module induces a module structure on the set of its fixpoints.

In a judgment $\Gamma \vdash e : \tau$, the coeffects in Γ describe an equivalence relation on $\mathrm{dom}(\Gamma) \cup \{\mathrm{R}\}$ where each coeffect corresponds to an equivalence class. The fact that two variables, say x and y, have the same coeffect means that

$$
\begin{array}{lcll}
\Gamma, \Delta & ::= & x_1 :_{X_1} \tau_1, \ldots, x_n :_{X_n} \tau_n & \text{(type-and-coeffect) context} \\
X & ::= & \{\ell_1, \ldots, \ell_n\} & \text{coeffect (set of links)}
\end{array}
$$

$$
\text{(T-VAR)}\ \frac{}{\emptyset \cdot \Gamma + x :_{\{\mathrm{R}\}} \tau \vdash x : \tau} \qquad \text{(T-CONST)}\ \frac{}{\emptyset \cdot \Gamma \vdash k : P_k}
$$

$$
\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash e : C \quad \mathsf{fields}(C) = \tau_1\, f_1; \ldots \tau_n\, f_n; \quad i \in 1..n}{\Gamma \vdash e.f_i : \tau_i}
$$

$$
\text{(T-FIELD-ASSIGN)}\ \frac{\Gamma \vdash e : C \quad \Delta \vdash e' : \tau_i \quad \mathsf{fields}(C) = \tau_1\, f_1; \ldots \tau_n\, f_n; \quad i \in 1..n}{\Gamma + \Delta \vdash e.f_i = e' : \tau_i}
$$

$$
\text{(T-NEW)}\ \frac{\Gamma_i \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Gamma_1 + \ldots + \Gamma_n \vdash \mathtt{new}\ C(e_1, \ldots, e_n) : C} \quad \mathsf{fields}(C) = \tau_1\, f_1; \ldots \tau_n\, f_n;
$$

$$
\text{(T-INVK)}\ \frac{\Gamma_0 \vdash e_0 : C \qquad \Gamma_i \vdash e_i : \tau_i \quad \forall i \in 1..n}{X_0' \cdot \Gamma_0 + \ldots + X_n' \cdot \Gamma_n \vdash e_0.m(e_1, \ldots, e_n) : \tau}
\begin{array}{l}
\mathsf{mtype}(C, m) \equiv^{\mathrm{fr}} X_0, \tau_1^{X_1} \ldots \tau_n^{X_n} \to \tau \\
\ell_0, \ldots, \ell_n \text{ fresh} \\
X_i' = X_i \cup \{\ell_i\} \quad \forall i \in 1..n
\end{array}
$$

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

$$
\text{(T-PRIM)}\ \frac{\Gamma \vdash e : P}{\{\ell\} \cdot \Gamma \vdash e : P}\ \ell \text{ fresh}
$$

$$
\text{(T-OBJ)}\ \frac{\Gamma_i \vdash v_i : \tau_i \quad \forall i \in 1..n}{\Gamma_1 + \cdots + \Gamma_n \vdash [v_1, \ldots, v_n]^C : C} \quad \mathsf{fields}(C) = \tau_1\, f_1; \ldots \tau_n\, f_n;
$$

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

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

FIGURE 6.3 Coeffect system for sharing

the evaluation of e possibly introduces sharing between x and y. Moreover, the fact that R is in the coeffect of x models possible sharing between x and the final result of the expression. Intuitively, sharing only happens among variables of reference types (classes), since a variable x of a primitive type P denotes an immutable value rather than a reference in memory. To have a uniform treatment, this is modeled by the fact that a judgment $x :_{\{\ell\}} P \vdash x : P$ with ℓ fresh is derivable (by rules (T-VAR) and (T-PRIM), as detailed below[2]).

The typing rules are given in Figure 6.3.

In the rule for variable, the variable is obviously linked with the result (they coincide), hence its coeffect is $\{\mathrm{R}\}$. In rule (T-CONST), no variable is used.

In rule (T-FIELD-ACCESS), the coeffects are those of the receiver expression, since the receiver is in sharing with its field. In rule (T-FIELD-ASSIGN), the cof-

2 Alternatively, variables of primitive types could be in a separate context, with no coeffects.

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

In rule (T-INVK), the coeffects of the arguments are summed, after multiplying each of them with the coeffect of the corresponding parameter, where, to avoid clashes, we assume that links different from R are freshly renamed, as indicated by the notation $\equiv^{\mathsf{fr}}$. Moreover, a fresh link ℓ_i is added[3], since otherwise, if the parameter is not used in the body (hence has empty coeffect), the links of the argument would be lost in the final context, see the example for rule (T-BLOCK) below.

The auxiliary function mtype is the same as in previous chapters. The condition that method bodies should conform to method types is expressed as follows:

(T-METH) $\mathsf{mtype}(C, m) = X_0, \tau_1^{X_1} \ldots \tau_n^{X_n} \to \tau$ implies
$\mathsf{mbody}(C, m) = \langle x_1 \ldots x_n, e \rangle$ and
$\mathtt{this} :_{X_0'} C, x_1 :_{X_1'} \tau_1, \ldots, x_n :_{X_n'} \tau_n \vdash e : \tau$
$X_i' = X_j' \neq \emptyset$ implies $X_i = X_j \neq \emptyset$

That is, coeffects computed by typechecking the method body should express no more sharing than those in the method type.

As an example, consider the following method:

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

where $\mathsf{mtype}(\mathtt{C},\mathtt{m})=\{\ell\}, \mathtt{B}^{\{\ell\}}, \mathtt{B}^{\{\mathtt{R}\}}, \mathtt{B}^{\{\mathtt{R}\}} \to \mathtt{C}$, with $\ell \neq \mathtt{R}$. The method body is well-typed, since we derive

$\mathtt{this} :_{\{\ell\}} \mathtt{C}, \mathtt{y} :_{\{\ell\}} \mathtt{B}, \mathtt{z1} :_{\{\mathtt{R}\}} \mathtt{B}, \mathtt{z2} :_{\{\mathtt{R}\}} \mathtt{B} \vdash \mathtt{x.f1=y;new\ C(z1, z2)} : \mathtt{C}$,
with $\ell \neq \mathtt{R}$

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

$$\dfrac{\dfrac{}{\mathtt{x} :_{\mathtt{C}} \{\mathtt{R}\} \vdash \mathtt{x} : \mathtt{C}}\,(\text{T-VAR}) \quad \dfrac{}{\mathtt{z} :_{\mathtt{B}} \{\mathtt{R}\} \vdash \mathtt{z} : \mathtt{B}}\,(\text{T-VAR}) \quad \dfrac{}{\mathtt{y_1} :_{\mathtt{B}} \{\mathtt{R}\} \vdash \mathtt{y_1} : \mathtt{B}}\,(\text{T-VAR}) \quad \dfrac{}{\mathtt{y_2} :_{\mathtt{B}} \{\mathtt{R}\} \vdash \mathtt{y_2} : \mathtt{B}}\,(\text{T-VAR})}{\mathtt{x} :_{\mathtt{C}} X, \mathtt{z} :_{\mathtt{B}} X, \mathtt{y_1} :_{\mathtt{B}} Y, \mathtt{y_2} :_{\mathtt{B}} Y \vdash \mathtt{x.m(z, y_1, y_2)} : \mathtt{C}}\,(\text{T-INVK})$$

where $X = \{\ell', \ell_0, \ell_1\}$ and $Y = \{\mathtt{R}, \ell_2, \ell_3\}$.
The context of the call is obtained as follows

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

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

3 Analogously to the rule (T-APP) in Figure 2.2 in the call-by-value case.

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

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

where $X = \{\ell', \ell_0, \ell_1, \ell_2, \ell_3, \mathtt{R}\}$. That is, `x`, `y`, `z`, and the result, are in sharing (note the role of the transitive closure here).
In the examples that follow we will omit the fresh links unless necessary.

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

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

$$\mathtt{z1} :_{\{\ell\}} \mathtt{B}, \mathtt{z2} :_{\{\ell\}} \mathtt{B} \vdash \mathtt{new\ C(z1,z2).f1.f} : \mathtt{int}, \text{ with } \ell \neq \mathtt{R}$$

shows that there is no longer a link between the result and `z1`, `z2`.

Rule (T-OBJ) is straightforward. In rule (T-MEM), a memory is well-formed in a context which is the sum of two parts. The former assigns a type to all and only references in memory, as in the standard rule in Figure 6.1, and a fresh singleton coeffect. The latter sums the coeffects of the objects in memory, after multiplying each of them with that of the corresponding reference.
For instance, for $\mathtt{x} \mapsto [\mathtt{y}]^{\mathtt{A}}, \mathtt{y} \mapsto [0]^{\mathtt{B}}, \mathtt{z} \mapsto [\mathtt{y}]^{\mathtt{A}}$, the former context is $\mathtt{x} :_{\{\ell_{\mathtt{x}}\}} \mathtt{A}, \mathtt{y} :_{\{\ell_{\mathtt{y}}\}} \mathtt{B}, \mathtt{z} :_{\{\ell_{\mathtt{z}}\}} \mathtt{A}$, the latter is the sum of the three contexts $\mathtt{y} :_{\{\ell_{\mathtt{x}}\}} \mathtt{A}$, $\emptyset$, and $\mathtt{y} :_{\{\ell_{\mathtt{z}}\}} \mathtt{A}$. Altogether, we get $\mathtt{x} :_{\{\ell_{\mathtt{x}},\ell_{\mathtt{y}},\ell_{\mathtt{z}}\}} \mathtt{A}, \mathtt{y} :_{\{\ell_{\mathtt{x}},\ell_{\mathtt{y}},\ell_{\mathtt{z}}\}} \mathtt{B}, \mathtt{z} :_{\{\ell_{\mathtt{x}},\ell_{\mathtt{y}},\ell_{\mathtt{z}}\}} \mathtt{A}$, expressing that the three references are connected. Note that no R link occurs in memory; indeed, there is no final result.

In rule (T-CONF), the coeffects of the expression and those of the memory are summed.

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

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

Here `x.f1=y;new C(z,z)` abbreviates `{B w = (x.f1=y); new C(z,z)}`. The derivation is in Figure 6.4, where the subderivations $\mathcal{D}_1$ and $\mathcal{D}_2$ are given separately for space reasons.

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash 2 : \texttt{int}}\,(\textsc{t-const})}{\emptyset \vdash \texttt{new B(2)} : \texttt{B}}\,(\textsc{t-new}) \qquad \dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Gamma \vdash \{\texttt{B w = (x.f1=y); new C(z,z)}\} : \texttt{C}}\,(\textsc{t-block})}{\texttt{x} :_{\{\ell\}} \texttt{C}, \texttt{y} :_{\{\ell\}} \texttt{B} \vdash \{\texttt{B z = new B(2); x.f1=y;new C(z,z)}\} : \texttt{C}}\,(\textsc{t-block})$$

ℓ, ℓ' fresh
$\texttt{x} :_{\{\ell\}} \texttt{C}, \texttt{y} :_{\{\ell\}} \texttt{B} = (\{\texttt{R}\} + \{\ell'\}) \cdot \emptyset + \texttt{x} :_{\{\ell\}} \texttt{C}, \texttt{y} :_{\{\ell\}} \texttt{B}$
$\Gamma = (\emptyset + \{\ell\}) \cdot (\texttt{x} :_{\{\texttt{R}\}} \texttt{C}, \texttt{y} :_{\{\texttt{R}\}} \texttt{B}) + \texttt{z} :_{\{\texttt{R}\}} \texttt{B} = \texttt{x} :_{\{\ell\}} \texttt{C}, \texttt{y} :_{\{\ell\}} \texttt{B}, \texttt{z} :_{\{\texttt{R}\}} \texttt{B}$

$$\mathcal{D}_1 = \dfrac{\dfrac{}{\texttt{x} :_{\{\texttt{R}\}} \texttt{C} \vdash \texttt{x} : \texttt{C}}\,(\textsc{t-var}) \qquad \dfrac{}{\texttt{y} :_{\{\texttt{R}\}} \texttt{B} \vdash \texttt{y} : \texttt{B}}\,(\textsc{t-var})}{\texttt{x} :_{\{\texttt{R}\}} \texttt{C}, \texttt{y} :_{\{\texttt{R}\}} \texttt{B} \vdash \texttt{x.f1=y} : \texttt{B}}\,(\textsc{t-field-assign})$$

$$\mathcal{D}_2 = \dfrac{\dfrac{}{\texttt{w} :_{\emptyset} \texttt{B}, \texttt{z} :_{\{\texttt{R}\}} \texttt{B} \vdash \texttt{z} : \texttt{B}}\,(\textsc{t-var}) \qquad \dfrac{}{\texttt{w} :_{\emptyset} \texttt{B}, \texttt{z} :_{\{\texttt{R}\}} \texttt{B} \vdash \texttt{z} : \texttt{B}}\,(\textsc{t-var})}{\texttt{w} :_{\emptyset} \texttt{B}, \texttt{z} :_{\{\texttt{R}\}} \texttt{B} \vdash \texttt{new C(z,z)} : \texttt{C}}\,(\textsc{t-new})$$

FIGURE 6.4 Example of type derivation

The rules in Figure 6.3 immediately lead to an algorithm which inductively computes the coeffects of an expression. Indeed, all the rules except (T-PRIM) are syntax-directed, that is, the coeffects of the expression in the consequence are computed as a linear combination of those of the subexpressions, where the basis is the rule for variables. Rule (T-PRIM) is assumed to be *always* used in the algorithm, just once, for expressions of primitive types.

We assume there are coeffect annotations in method parameters to handle (mutual) recursion; for non-recursive methods, such coeffects can be computed (that is, in the coherency condition above, the X_is in mtype are exactly the X_i's). We leave to future work the investigation of a global fixed-point inference to compute coeffects across mutually recursive methods (some care is needed to ensure termination since we generate fresh links for the calls).

Considering again Example 6.2.2:

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

The parts emphasized in gray are the coeffects which can be computed for the parameters by typechecking the body (the coeffect for `this` is in square brackets). In a real language, such coeffects would be declared by some concrete syntax, as part of the type information available to clients. From such coeffects, a client knows that the result of `mix` will be connected to both the receiver

and the argument, whereas the result of `clone` will be a reference to a *fresh* portion of memory, not connected to the receiver.

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

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

expressing that `a1` is linked to the result.

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

Given $\Gamma = x_1 :_{X_1} \tau_1, \ldots, x_n :_{X_n} \tau_n$, set $\mathsf{coeff}(\Gamma, x_i) = X_i$ and $\mathsf{links}(\Gamma) = \bigcup_{i \in 1..n} X_i \cup \{\mathrm{R}\}$. Moreover, the *restriction* of Γ to the set of variables $V = \{x_1, \ldots, x_m\}$, with $m \leq n$, and the set of links X, denoted $\Gamma \restriction \langle V, X \rangle$, is the context $x_1 :_{Y_1} \tau_1, \ldots, x_m :_{Y_m} \tau_m$ where, for each $i \in 1..m$, $Y_i = X_i \cap X$. In the following, $\Gamma \restriction \Delta$ abbreviates $\Gamma \restriction \langle \mathsf{dom}(\Delta), \mathsf{links}(\Delta) \rangle$.

Recall that $\bowtie_\mu$ denotes the sharing relation in memory μ (Definition 6.2.1). The following result shows that the typing of the memory precisely captures the sharing relation.

LEMMA 6.3.2 : If $\Gamma \vdash \mu$, then $x \bowtie_\mu y$ if and only if $\mathsf{coeff}(\Gamma, x) = \mathsf{coeff}(\Gamma, y)$.

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

THEOREM 6.3.3 (Subject reduction): If $\Gamma \vdash e|\mu : \tau$ and $\langle e, \mu \rangle \longrightarrow \langle e', \mu' \rangle$, then $\Delta \vdash e'|\mu' : \tau$, for some Δ such that $(\Gamma + \Delta) \restriction \Gamma = \Gamma$.

To prove this theorem we need some auxiliary definitions and lemmas.

DEFINITION 6.3.4 : $\Gamma \blacktriangleleft \Gamma'$ if

1. $\Gamma' = \Gamma$ or
2. $\Gamma' = \{\ell\} \cdot \Gamma$ with ℓ fresh

LEMMA 6.3.5 : If $\mathcal{D} : \Gamma \vdash e : \tau$, then there is a subderivation $\mathcal{D}' : \Gamma' \vdash e : \tau$ of $\mathcal{D}$ ending with a syntax-directed rule and $\Gamma' \blacktriangleleft \Gamma$

Proof: By induction on $\mathcal{D} : \Gamma \vdash e : \tau$

if $\mathcal{D}$ ends with a syntax-directed rule then $\Gamma' = \Gamma$ and $\mathcal{D} = \mathcal{D}'$. If $\mathcal{D}$ ends with rule (T-PRIM) we have $\Gamma = \{\ell\} \cdot \Delta$ with ℓ fresh and so $\Delta \blacktriangleleft \Gamma$ where $\Delta \vdash e : \tau$. By induction hypothesis on the premise we have a subderivation $\mathcal{D}' : \Gamma' \vdash e : \tau$ ending with a syntax-directed rule and $\Gamma' \blacktriangleleft \Delta$. We have two cases:

- $\Gamma' = \Delta$: by this and by $\Delta \blacktriangleleft \Gamma$ we obtain $\Gamma' \blacktriangleleft \Gamma$, that is, the thesis
- $\Delta = \{\ell'\} \cdot \Gamma'$ with ℓ' fresh: by this and $\Gamma = \{\ell\} \cdot \Delta$ we obtain $\Gamma = \{\ell\} \cdot \{\ell'\} \cdot \Gamma' = \{\ell'\} \cdot \Gamma'$ so we have $\Gamma' \blacktriangleleft \Gamma$ and so the thesis

□

LEMMA 6.3.6 (Inversion): If $\Gamma \vdash e : \tau$ then exists a context Γ' such that $\Gamma' \blacktriangleleft \Gamma$ and $\Gamma' \vdash e : \tau$ and the following properties holds:

1. If $e = x$ then $\Gamma' = \emptyset \cdot \Gamma'' + x :_{\{\mathrm{R}\}} \tau$
2. If $e = k$, then $\tau = P_k$.
3. If $e = e\,.\,f_i$, then $\Gamma' \vdash e : C$ and $\mathsf{fields}(C) = \tau_1\ f_1\mathtt{;}\ \ldots \tau_n\ f_n\mathtt{;}$ and $\tau_i = \tau$.
4. If $e = e\,.\,f_i = e'$ then $\Gamma_1 \vdash e : C$ and $\mathsf{fields}(C) = \tau_1\ f_1\mathtt{;}\ \ldots \tau_n\ f_n\mathtt{;}$ and $\tau_i = \tau$ and $\Gamma_2 \vdash e' : \tau_i$ with $\Gamma' = \Gamma_1 + \Gamma_2$.
5. If $e = \mathtt{new}\ C(e_1, \ldots, e_n)C$, then we have $\mathsf{fields}(C) = \tau_1\ f_1\mathtt{;}\ \ldots \tau_n\ f_n\mathtt{;}$ and $\Gamma_i \vdash e_i : \tau_i$ for all $i \in 1..n$ and $\Gamma = \Gamma_1 + \ldots + \Gamma_n$.
6. If $e = e_0.m(e_1, \ldots, e_n)$, then $\Gamma_0 \vdash e_0 : C$ and $\mathsf{mtype}(C, m) = X_{\mathtt{this}}, \tau_1^{X_1} \ldots \ldots \tau_n^{X_n} \to \tau$ and $\Gamma_i \vdash e_i : \tau_i$ for all $i \in 1..n$ and $\Gamma' = (X_{\mathtt{this}} \cdot \Gamma_0) + (X_1 \cdot \Gamma_1) + \ldots + (X_n \cdot \Gamma_n)$.
7. If $e = \{\tau\ x = e\mathtt{;}\ e'\}$ then $\Gamma' = (X + \{\ell\}) \cdot \Gamma_1 + \Gamma_2$ where ℓ is fresh and $\Gamma_1 \vdash e : \tau$ and $\Gamma_2, x :_X \tau' \vdash e' : \tau'$.

Proof:

1. If $\mathcal{D} : \Gamma \vdash x : \tau$ then, by Lemma 6.3.5, we know that exists a derivation $\mathcal{D}' : \Gamma' \vdash x : \tau$ subderivation of $\mathcal{D}$ ending with a syntax-directed rule and $\Gamma' \blacktriangleleft \Gamma$. Since the last applied rule in $\mathcal{D}'$ must be (T-VAR), we have $\Gamma' = \emptyset \cdot \Gamma'' + x :_{\{\mathrm{R}\}} \tau \mathrel{\hat{\subseteq}} x :_{\{\mathrm{R}\}} \tau$
2. If $\Gamma \vdash k : \tau$ then the last applied rule can be (T-CONST) or (T-PRIM). In both cases we have the thesis
3. If $\mathcal{D} : \Gamma \vdash e\,.\,f_i : \tau$ then, by Lemma 6.3.5, we know that exists a derivation $\mathcal{D}' : \Gamma' \vdash e\,.\,f_i : \tau$ subderivation of $\mathcal{D}$ ending with a syntax-directed rule and $\Gamma' \blacktriangleleft \Gamma$. Since the last applied rule in $\mathcal{D}'$ must be (T-ACCESS) we have that $\Gamma' \vdash e : C$ with $\mathsf{fields}(C) = \tau_1\ f_1\mathtt{;}\ \ldots \tau_n\ f_n\mathtt{;}$ and $\tau_i = \tau$
4. If $\mathcal{D} : \Gamma \vdash e\,.\,f_i = e' : \tau$ then, by Lemma 6.3.5, we know that exists a derivation $\mathcal{D}' : \Gamma' \vdash e\,.\,f_i = e' : \tau$ subderivation of $\mathcal{D}$ ending with a syntax-directed rule and $\Gamma' \blacktriangleleft \Gamma$. Since the last applied rule in $\mathcal{D}'$ must be (T-ASSIGN), we have that $\Gamma' = \Delta_1 + \Delta_2$ such that $\Delta_1 \vdash e : C$ and $\Delta_2 \vdash e' : \tau_i$ with $\mathsf{fields}(C) = \tau_1\ f_1\mathtt{;}\ \ldots \tau_n\ f_n\mathtt{;}$ and $\tau_i = \tau$
5. If $\mathcal{D} : \Gamma \vdash \mathtt{new}\ C(e_1, \ldots, e_n) : C$ then, since the last applied rule in $\mathcal{D}$ must be (T-NEW), we have that $\Gamma = \Delta_1 + \ldots + \Delta_n$ such that $\mathsf{fields}(C) = \tau_1\ f_1\mathtt{;}\ \ldots \tau_n\ f_n\mathtt{;}$ and $\Delta_i \vdash e_i : \tau_i$ for all $i \in 1..n$.
6. If $\mathcal{D} : \Gamma \vdash e_0.m(e_1, \ldots, e_n) : \tau$ then, by Lemma 6.3.5, we know that

exists a derivation $\mathcal{D}' : \Gamma' \vdash e_0.m(e_1, \ldots, e_n) : \tau$ subderivation of $\mathcal{D}$ ending with a syntax-directed rule and $\Gamma' \blacktriangleleft \Gamma$. Since the last applied rule in $\mathcal{D}'$ must be (T-INVK), we have $\Delta_0 \vdash e_0 : C$, $\mathrm{mtype}(C, m) = X_{\mathtt{this}}, \tau_1^{X_1} \ldots \tau_n^{X_n} \to \tau$ and $\Delta_i \vdash e_i : \tau_i$ for all $i \in 1..n$ and $\Gamma' = (X_{\mathtt{this}} \cdot \Delta_0) + (X_1 \cdot \Delta_1) + \ldots + (X_n \cdot \Delta_n)$.

7. If $\mathcal{D} : \Gamma \vdash \{\tau\ x = e;\ e'\} : \tau'$ then, by Lemma 6.3.5, we know that exists a derivation $\mathcal{D}' : \Gamma' \vdash \{\tau\ x = e;\ e'\} : \tau$ subderivation of $\mathcal{D}$ ending with a syntax-directed rule and $\Gamma' \blacktriangleleft \Gamma$. Since the last applied rule in $\mathcal{D}'$ must be (T-BLOCK), we have $\Gamma' = (X + \{\ell\}) \cdot \Delta' + \Gamma''$ where ℓ is fresh $\Delta' \vdash e : \tau$ and $\Gamma'', x :_X \tau' \vdash e' : \tau'$.

□

LEMMA 6.3.7 : If Γ is a closed context then, for all $x, y \in \mathrm{dom}(\Gamma)$, $\mathrm{coeff}(\Gamma, x) \cap \mathrm{coeff}(\Gamma, y) \neq \emptyset$ implies $\mathrm{coeff}(\Gamma, x) = \mathrm{coeff}(\Gamma, y)$.

Proof: Immediate from the definition of $\_^\star$. □

LEMMA 6.3.8 : Let Γ be a closed context and $\Delta = \Gamma, x :_X \tau$. Then, for all $y \in \mathrm{dom}(\Delta)$, we have

$$\mathrm{coeff}(\Delta^\star, y) = \begin{cases} \bigcup\{\mathrm{coeff}(\Delta, z) \mid \mathrm{coeff}(\Delta, z) \cap X \neq \emptyset\} & \mathrm{coeff}(\Delta, y) \cap X \neq \emptyset \\ \mathrm{coeff}(\Delta, y) & \text{otherwise} \end{cases}$$

Proof: Suppose $\Gamma = x_1 :_{X_1} \tau_1, \ldots, x_n :_{X_n} \tau_n$ with $n \geq 0$, $x :_X \tau = x_{n+1} :_{X_{n+1}} \tau_{n+1}$ and set $\Theta = x_1 :_{Y_1} \tau_1, \ldots, x_{n+1} :_{Y_{n+1}} \tau_{n+1}$, where for all $i \in 1..n+1$, we have

$$Y_i = \begin{cases} \bigcup\{X_j \mid j \in 1..n+1, X_j \cap X_{n+1} \neq \emptyset\} & X_i \cap X_{n+1} \neq \emptyset \\ X_i & \text{otherwise} \end{cases}$$

The inequality $\Theta \hat{\subseteq} \Delta^\star$ is trivial by definition of $\_^\star$. We know that $\Delta \subseteq \Theta$, so to get the other direction, we just have to show that Θ is closed. To this end, let $i \in 1..n+1$, $\ell_1 \in Y_i$ and $\ell_1, \ell_2 \in Y_j$ for some $j \in 1..n+1$, then we have to prove that $\ell_2 \in Y_i$. We distinguish to cases.

- If $X_j \cap X_{n+1} = \emptyset$, then $Y_j = X_j$. We observe that $\ell_1 \notin X_k$ for any $k \in 1..n+1$ such that $X_k \cap X_{n+1} \neq \emptyset$. This is obvious for $k = n+1$, as it is against the assumption $X_j \cap X_{n+1} = \emptyset$. For $k \in 1..n$, since Γ is closed, by Lemma 6.3.7, we would bet $X_j = X_k$, and so $X_j \cap X_{n+1} \neq \emptyset$, which is again a contraddiction. This implies that $X_i \cap X_{n+1} = \emptyset$ and so $Y_i = X_i$. Therefore, applying again Lemma 6.3.7, we get $Y_i = X_i = X_j = Y_j$, thus $\ell_2 \in Y_i$, as needed.
- If $X_j \cap X_{n+1} \neq \emptyset$, then we have $Y_j = \bigcup\{X_k \mid k \in 1..n+1, X_k \cap X_{n+1} \neq \emptyset\}$, hence $\ell_1 \in X_h$ for some $h \in 1..n+1$ such that $X_h \cap X_{n+1} \neq \emptyset$. By a

argument similar to the previous point, we get that $X_i \cap X_{n+1} \neq \emptyset$, hence, by definition, $Y_i = Y_j$ that proves the thesis.

□

LEMMA 6.3.9 : [Inversion for context] If $\Gamma \vdash \mathcal{E}[e] : \tau$, then, for some Γ', Δ, $x \notin \mathrm{dom}(\Gamma)$, X and τ', we have $\Gamma = (X \cdot \Delta) + \Gamma'$and $\Gamma' + x :_X \tau' \vdash \mathcal{E}[x] : \tau$ and $\Delta \vdash e : \tau'$.

Proof: The proof is by induction on $\mathcal{E}$. We only show some cases, the others are analogous.

$\mathcal{E} = [\ ]$ Just take $\Gamma' = \emptyset$, $x \notin \mathrm{dom}(\Gamma)$, $X = \{\mathrm{R}\}$, $\tau' = \tau$ and $\Delta = \Gamma$.

$\mathcal{E} = \mathcal{E}' \,.\, f = e'$ By Lemma 6.3.6(4) we have a context Γ' such that $\Gamma' \blacktriangleleft \Gamma$, $\Gamma_1 + \Gamma_2 = \Gamma'$, $\Gamma_1 \vdash \mathcal{E}'[e] : C$ and $\Gamma_2 \vdash e' : \tau$. By induction hypothesis we know that for some Γ'', Δ, $x \notin \mathrm{dom}(\Gamma)$, X and τ', $\Gamma_1 = (X \cdot \Delta) + \Gamma''$, $\Gamma'' + x :_X \tau' \vdash \mathcal{E}'[x] : C$ and $\Delta \vdash e : \tau'$. We can assume $x \notin \mathrm{dom}(\Gamma)$ (doing a step of renaming if needed). We get $\Gamma_1 + \Gamma_2 = (X \cdot \Delta) + \Gamma'' + \Gamma_2 = \Gamma'$ and $(\Gamma_2 + \Gamma'') + x :_X \tau' \vdash \mathcal{E}'[x] \,.\, f = e' : \tau$. We have two cases:

- $\tau = C$ We know that the last rule applied to derive $\Gamma \vdash \mathcal{E}[e] : \tau$ is (T-ASSIGN), so $\Gamma = \Gamma_1 + \Gamma_2 = \Gamma'$, so we have the thesis
- $\tau = P$ We have two cases. If $\Gamma = \Gamma'$ we have the same situation as above so we have the thesis. If $\Gamma = \{\ell\} \cdot \Gamma'$ then we have $\Gamma = \{\ell\} \cdot ((X \cdot \Delta) + \Gamma'' + \Gamma_2) = (\{\ell\} \cdot X) \cdot \Delta) + \{\ell\} \cdot (\Gamma'' + \Gamma_2)$. We can apply rule (T-PRIM) to $(\Gamma_2 + \Gamma'') + x :_X \tau' \vdash \mathcal{E}'[x] \,.\, f = e' : \tau$ to obtain $\{\ell\} \cdot ((\Gamma_2 + \Gamma'') + x :_X \tau') \vdash \mathcal{E}'[x] \,.\, f = e' : \tau$. We know $\{\ell\} \cdot ((\Gamma_2 + \Gamma'') + x :_X \tau') = \{\ell\} \cdot (\Gamma_2 + \Gamma'') + x :_{\{\ell\} \cdot X} \tau'$.

$\mathcal{E} = \{\tau'\ x = \mathcal{E}'\ ;\ e'\}$ By Lemma 6.3.6(7) we have a context Γ' such that $\Gamma' \blacktriangleleft \Gamma$,

$\Gamma' = (X + \{\ell\}) \cdot \Gamma_1 + \Gamma_2$ where ℓ is fresh and $\Gamma_1 \vdash \mathcal{E}'[e] : \tau$ and $\Gamma_2, x :_X \tau' \vdash e' : \tau'$.

By induction hypothesis we know that for some Γ'', Δ, $y \notin \mathrm{dom}(\Gamma)$, Y and τ', $\Gamma_1 = (Y \cdot \Delta) + \Gamma''$, $\Gamma'' + y :_Y \tau' \vdash \mathcal{E}'[y] : C$ and $\Delta \vdash e : \tau'$. We can assume $y \notin \mathrm{dom}(\Gamma)$ (doing a step of renaming if needed). We get $(X + \{\ell\}) \cdot \Gamma_1 + \Gamma_2 = (X + \{\ell\}) \cdot ((Y \cdot \Delta) + \Gamma'') + \Gamma_2 = (((X + \{\ell\}) \cdot Y) \cdot \Delta) + ((X + \{\ell\}) \cdot \Gamma'') + \Gamma_2 = \Gamma'$. By rule (T-BLOCK) we have $((X + \{\ell\}) \cdot \Gamma'') + \Gamma_2 + y :_{((X+\{\ell\}) \cdot Y)} \tau' \vdash \mathcal{E}[y] : \tau$. We have two cases:

- $\tau = C$ We know that the last rule applied to derive $\Gamma \vdash \mathcal{E}[e] : \tau$ is (T-BLOCK), so $\Gamma = (X + \{\ell\}) \cdot \Gamma_1 + \Gamma_2 = \Gamma'$, so we have the thesis
- $\tau = P$ We have two cases. If $\Gamma = \Gamma'$ we have the same situation as above so we have the thesis. If $\Gamma = \{\ell'\} \cdot \Gamma'$ then we have $\Gamma = \{\ell'\} \cdot ((X + \{\ell\}) \cdot \Gamma_1 + \Gamma_2) = (\{\ell'\} \cdot (X + \{\ell\}) \cdot \Gamma_1) + \{\ell'\} \cdot \Gamma_2 = ((((\{\ell'\} \cdot (X + \{\ell\}) \cdot Y) \cdot \Delta) + ((\{\ell'\} \cdot (X + \{\ell\}) \cdot \Gamma'') + \{\ell'\} \cdot \Gamma_2$. We can apply rule (T-PRIM) to $((X + \{\ell\}) \cdot \Gamma'') + \Gamma_2 + y :_{((X+\{\ell\}) \cdot Y)} \tau' \vdash \mathcal{E}[y] : \tau$ to obtain $\{\ell'\} \cdot (((X + \{\ell\}) \cdot \Gamma'') + \Gamma_2 + y :_{\tau'} ((X + \{\ell\}) \cdot Y)) \vdash \mathcal{E}[y] : \tau$.

We know $\{\ell'\} \cdot (((X + \{\ell\}) \cdot \Gamma'') + \Gamma_2 + y :_{((X+\{\ell\})\cdot Y)} \tau') = (\{\ell'\} \cdot (X + \{\ell\}) \cdot \Gamma'') + \{\ell'\} \cdot \Gamma_2 + y :_{(\{\ell'\}\cdot(X+\{\ell\})\cdot Y)} \tau'$ By $(\{\ell'\} \cdot (X + \{\ell\}) \cdot \Gamma'') + \{\ell'\} \cdot \Gamma_2 + y :_{(\{\ell'\}\cdot(X+\{\ell\})\cdot Y)} \tau' \vdash \mathcal{E}[y] : \tau$ we obtain the thesis.

□

LEMMA 6.3.10 : If $\Delta \vdash e : \tau$ and $\Gamma + x :_X \tau \vdash \mathcal{E}[x] : \tau'$, with $x \notin \mathrm{dom}(\Gamma)$, then $\Gamma + X \cdot \Delta \vdash \mathcal{E}[e] : \tau'$.

Proof: By induction on the derivation of $\Gamma + x :_X T \vdash \mathcal{E}[x] : T'$. □

LEMMA 6.3.11 : Let Γ be a closed context and $\Delta = \sum_{x \in \mathrm{dom}(\Delta)} x :_X \tau_x$ and such that $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma)$ and $\mathsf{links}(\Gamma) \cap \mathsf{links}(\Delta) = \{\mathrm{R}\}$. We have two cases:

- $\mathrm{R} \notin X$
  For all $y \in \mathrm{dom}(\Gamma)$, if exists $x \in \mathrm{dom}(\Delta)$ such that $\mathsf{coeff}(\Gamma, y) \cap \mathsf{coeff}(\Gamma, x) \neq \emptyset$ then $\mathsf{coeff}(\Gamma + \Delta, y) = X + \sum_{x \in \Delta} \mathsf{coeff}(\Gamma, x)$, otherwise $\mathsf{coeff}(\Gamma + \Delta, y) = \mathsf{coeff}(\Gamma, y)$.
- $\mathrm{R} \in X$
  We define $Y = \mathsf{coeff}(\Gamma, z)$ if $\mathrm{R} \in \mathsf{coeff}(\Gamma, z)$ and $z \in \mathrm{dom}(\Gamma)$. For all $y \in \mathrm{dom}(\Gamma)$, if exists $x \in \mathrm{dom}(\Delta)$ such that $\mathsf{coeff}(\Gamma, y) \cap \mathsf{coeff}(\Gamma, x) \neq \emptyset$ or $\mathrm{R} \in \mathsf{coeff}(\Gamma, y)$ then $\mathsf{coeff}(\Gamma + \Delta, y) = X + \sum_{x \in \Delta} \mathsf{coeff}(\Gamma, x) + Y$, otherwise $\mathsf{coeff}(\Gamma + \Delta, y) = \mathsf{coeff}(\Gamma, y)$.

LEMMA 6.3.12 : Let $\Gamma \upharpoonright \Delta = \Delta$.

1. $(X \cdot \Gamma) \upharpoonright (X \cdot \Delta) = X \cdot \Delta$
2. If $\mathsf{links}(\Theta) \cap (\mathsf{links}(\Delta) \cup \mathsf{links}(\Gamma)) = \{\mathrm{R}\}$ or $\mathsf{links}(\Theta) \cap (\mathsf{links}(\Delta) \cup \mathsf{links}(\Gamma)) = \{\mathrm{R}\} \cup X$ where exists $x \in \mathrm{dom}(\Delta)$ such that $\mathsf{coeff}(\Delta, x) = X$ and $\mathrm{dom}(\Theta) \subseteq \mathrm{dom}(\Delta)$, then $(\Gamma + \Theta) \upharpoonright (\Delta + \Theta) = \Delta + \Theta$.

Proof: 1. Let $x \in \mathrm{dom}(\Delta)$ and $\mathsf{coeff}(\Gamma, x) = X_\Gamma$ and $\mathsf{coeff}(\Delta, x) = X_\Delta$. By definition of $\Gamma \upharpoonright \Delta$ and $\Gamma \upharpoonright \Delta = \Delta$, we have that $X_\Gamma \cap \mathsf{links}(\Delta) = X_\Delta$. Let Y be such that $\mathsf{coeff}(X \cdot \Gamma, x) = Y$, then $Y = X \triangleleft X_\Gamma$. We want to prove that $Y \cap \mathsf{links}(X \triangleleft \Delta) = X \triangleleft X_\Delta$. Consider two cases: $\mathrm{R} \notin X_\Gamma$ and $\mathrm{R} \in X_\Gamma$. In the first case $Y = X_\Gamma$ and $Y \cap (\mathsf{links}(X \triangleleft \Delta)) = Y \cap \mathsf{links}(\Delta) = X_\Delta$. Moreover, since $\mathrm{R} \notin X_\Delta$ we have $X \triangleleft X_\Delta = X_\Delta$. Therefore, $Y \cap (\mathsf{links}(X \triangleleft \Delta)) = X \triangleleft X_\Delta$.
In the second case $Y = X \cup (X_\Gamma - \{\mathrm{R}\})$ and since $\mathrm{R} \in X_\Delta$ we have $X \triangleleft X_\Delta = X \cup (X_\Delta - \{\mathrm{R}\})$. From $X_\Gamma \cap \mathsf{links}(\Delta) = X_\Delta$ we get $(X_\Gamma - \{\mathrm{R}\}) \cap \mathsf{links}(\Delta) = X_\Delta - \{\mathrm{R}\}$. Note that $X \subseteq \mathsf{links}(X \triangleleft \Delta)$. Therefore $Y \cap (\mathsf{links}(X \triangleleft \Delta)) = X \cup (X_\Delta - \{\mathrm{R}\})$ which proves the result.

2. We know $\Theta = \hat{\cup}_{i=0}^{n} \Theta_i$, where for all $i \in [1...n]$, in Θ_i all variables have the same coeffect X_i and, for all $j, k \in [1...n]$ and $j \neq k$, $\mathsf{links}(\Theta_j) \cap \mathsf{links}(\Theta_k) = \{\mathrm{R}\}$. To obtain the thesis therefore we just need to prove that the property holds summing one Θ_i at a time, since $\Gamma + \Theta$ and

$\Delta + \Theta$ can be obtained summing iteratively the Θ_is. It suffices to prove the thesis only for the first sum. By $\Gamma \restriction \Delta = \Delta$ we have that for all $x \in \mathsf{dom}(\Delta)$, $\mathsf{coeff}(\Gamma, x) = Z \cup \mathsf{coeff}(\Delta, x)$, where $Z \cap \mathsf{links}(\Delta) = \emptyset$. By Lemma 6.3.11 we have two cases for $\Gamma + \Theta_i$ and $\Delta + \Theta_i$:

- $\mathrm{R} \notin X_i$
  For all $y \in \mathsf{dom}(\Gamma)$, if exists $x \in \mathsf{dom}(\Theta_i)$ such that $\mathsf{coeff}(\Gamma, y) \cap \mathsf{coeff}(\Gamma, x) \neq \emptyset$ then $\mathsf{coeff}(\Gamma + \Theta_i, y) = X + \sum_{x \in \Theta_i} \mathsf{coeff}(\Gamma, x) = X + \sum_{x \in \Theta_i} (\mathsf{coeff}(\Delta, x) \cup Z_x)$, otherwise $\mathsf{coeff}(\Gamma + \Theta_i, y) = \mathsf{coeff}(\Gamma, y) = Z_y \cup \mathsf{coeff}(\Delta, y)$. For all $y \in \mathsf{dom}(\Delta)$, if exists $x \in \mathsf{dom}(\Theta_i)$ such that $\mathsf{coeff}(\Delta, y) \cap \mathsf{coeff}(\Delta, x) \neq \emptyset$ then $\mathsf{coeff}(\Delta + \Theta_i, y) = X + \sum_{x \in \Theta_i} \mathsf{coeff}(\Delta, x)$, otherwise $\mathsf{coeff}(\Delta + \Theta_i, y) = \mathsf{coeff}(\Delta, y)$. By $\Gamma \restriction \Delta = \Delta$ we have that, for all $x, y \in \mathsf{dom}(\Delta)$, $\mathsf{coeff}(\Gamma, x) \cap \mathsf{coeff}(\Gamma, y) \neq \emptyset$ if and only if $\mathsf{coeff}(\Delta, x) \cap \mathsf{coeff}(\Delta, y) \neq \emptyset$. By these considerations we derive that, for all $x \in \mathsf{dom}(\Gamma)$, $\mathsf{coeff}(\Gamma + \Theta_i, x) \cap \mathsf{links}(\Delta + \Theta_i) = \mathsf{coeff}(\Delta + \Theta_i,)$, that is, $\Gamma + \Theta_i \restriction \Delta + \Theta_i = \Delta + \Theta_i$.
- $\mathrm{R} \in X_i$
  We define $Y = \mathsf{coeff}(\Gamma, z)$ if $\mathrm{R} \in \mathsf{coeff}(\Gamma, z)$ and $z \in \mathsf{dom}(\Gamma)$. For all $y \in \mathsf{dom}(\Gamma)$, if exists $x \in \mathsf{dom}(\Theta_i)$ such that $\mathsf{coeff}(\Gamma, y) \cap \mathsf{coeff}(\Gamma, x) \neq \emptyset$ or $\mathrm{R} \in \mathsf{coeff}(\Gamma, y)$ then $\mathsf{coeff}(\Gamma + \Theta_i, y) = X \cup \sum_{x \in \Theta_i} \mathsf{coeff}(\Gamma, x) \cup Y = X \cup \sum_{x \in \Theta_i} (\mathsf{coeff}(\Delta, x) \cup Z_x) \cup Y$, otherwise $\mathsf{coeff}(\Gamma + \Theta_i, y) = \mathsf{coeff}(\Gamma, y) = Z_y \cup \mathsf{coeff}(\Delta, y)$. We define $Y = \mathsf{coeff}(\Delta, z)$ if $\mathrm{R} \in \mathsf{coeff}(\Delta, z)$ and $z \in \mathsf{dom}(\Delta)$. For all $y \in \mathsf{dom}(\Delta)$, if exists $x \in \mathsf{dom}(\Theta_i)$ such that $\mathsf{coeff}(\Delta, y) \cap \mathsf{coeff}(\Delta, x) \neq \emptyset$ or $\mathrm{R} \in \mathsf{coeff}(\Theta_i, y)$ then $\mathsf{coeff}(\Delta + \Theta_i, y) = X + \sum_{x \in \Theta_i} \mathsf{coeff}(\Delta, x) + Y$, otherwise $\mathsf{coeff}(\Gamma + \Theta_i, y) = \mathsf{coeff}(\Gamma, y)$. By $\Gamma \restriction \Delta = \Delta$ we have that, for all $x, y \in \mathsf{dom}(\Delta)$, $\mathsf{coeff}(\Gamma, x) \cap \mathsf{coeff}(\Gamma, y) \neq \emptyset$ if and only if $\mathsf{coeff}(\Delta, x) \cap \mathsf{coeff}(\Delta, y) \neq \emptyset$. By these considerations we derive that, for all $x \in \mathsf{dom}(\Gamma)$, $\mathsf{coeff}(\Gamma + \Theta_i, x) \cap \mathsf{links}(\Delta + \Theta_i) = \mathsf{coeff}(\Delta + \Theta_i,)$, that is, $\Gamma + \Theta_i \restriction \Delta + \Theta_i = \Delta + \Theta_i$.

□

LEMMA 6.3.13 (Substitution): If $\Delta \vdash e' : \tau'$ and $\Gamma, x :_X \tau' \vdash e : \tau$ and Δ' is Δ with fresh renamed links, then $\Gamma' \vdash e[e'/x] : \tau$ with Γ' such that $\Gamma' \hat{\subseteq} X \triangleleft \Delta' + \Gamma$.

Proof: By induction on the derivation of $\Gamma, x :_X \tau' \vdash e : \tau$

(T-VAR) We know that $\Gamma = \emptyset$ and $X = \{\mathrm{R}\}$, hence the thesis follows from the hypothesis $\Delta \vdash e' : \tau'$, as $\emptyset + \{\mathrm{R}\} \cdot \Delta = \Delta$ and $x[e'/x] = e'$.

(T-INVK) We know that $e = e_0.m(e_1, \ldots, e_n)$, $\Gamma \hat{\subseteq} +_{0 \leq i \leq n} X_i \cdot \Gamma_i$ and $X \supseteq \cup_{0 \leq i \leq n} X_i \triangleleft X_i'$, where $\Gamma_i, x :_{X_i'} \tau' \vdash e_i : C_i$ $(0 \leq i \leq n)$ and $\mathsf{mtype}(C_0, m) = X_0, \tau_1^{X_1} \ldots \tau_n^{X_n} \to \tau$. By induction hypothesis we obtain $\Gamma_i + (X_i' \cdot \Delta') \vdash e_i[e/x] : \tau_i$ $(0 \leq i \leq n)$. Applying rule (T-INVK) we obtain

$$+_{0\le i\le n} X_i \cdot (\Gamma_i + X_i' \cdot \Delta') \vdash e_0.m(e_1,\ldots,e_n)[e/x] : \tau$$

Moreover, $+_{0\le i\le n} X_i \cdot (\Gamma_i + X_i' \cdot \Delta') = +_{0\le i\le n}(X_i \cdot \Gamma_i) + (X_i \triangleleft X_i') \cdot \Delta'$ and $+_{0\le i\le n}(X_i \triangleleft X_i') \cdot \Delta' = (\cup_{0\le i\le n} X_i \triangleleft X_i') \cdot \Delta'$.

We want to show that $+_{0\le i\le n}(X_i\cdot\Gamma_i)+(\cup_{0\le i\le n}X_i\triangleleft X_i'\cdot\Delta') \mathrel{\hat{\subseteq}} X \triangleleft \Delta'+\Gamma$. Knowing that $\Gamma \mathrel{\hat{\subseteq}} +_{0\le i\le n} X_i \cdot \Gamma_i$ and $X \supseteq \cup_{0\le i\le n} X_i \triangleleft X_i'$ we derive $+_{0\le i\le n}(X_i \cdot \Gamma_i) + (\cup_{0\le i\le n} X_i \triangleleft X_i' \cdot \Delta') \mathrel{\hat{\subseteq}} X \triangleleft \Delta' + \Gamma$.

(T-NEW) We know that $e = \texttt{new}\ C(e_1,\ldots,e_n)$, $\Gamma \mathrel{\hat{\subseteq}} +_{1\le i\le n}\Gamma_i$ and $X \supseteq \cup_{1\le i\le n} X_i$ and $\Gamma_i, x :_{X_i} \tau' \vdash e_i : \tau_i$ $(1 \le i \le n)$. By inductive hypothesis $\Gamma_i + (X_i \cdot \Delta') \vdash e_i[e'/x] : \tau_i$, $(1 \le i \le n)$. Applying rule (T-NEW) we obtain $+_{1\le i\le n}\Gamma_i + (X_i \cdot \Delta') \vdash \texttt{new}\ C(e_1,\ldots,e_n)\,[e'/x] : \tau_1$.

We know $+_{1\le i\le n}\Gamma_i+(X_i\cdot\Delta') = +_{1\le i\le n}\Gamma_i{+}{+}_{1\le i\le n}(X_i\cdot\Delta') = +_{1\le i\le n}\Gamma_i+ (\cup_{1\le i\le n}X_i \cdot \Delta')$.

We want to show that $+_{1\le i\le n}\Gamma_i + (\cup_{1\le i\le n}X_i \cdot \Delta') \mathrel{\hat{\subseteq}} X \triangleleft \Delta' + \Gamma$. By $X \supseteq \cup_{1\le i\le n}X_i$ and $\Gamma \mathrel{\hat{\subseteq}} +_{1\le i\le n}\Gamma_i$ we derive $X \triangleleft \Delta' + \Gamma \mathrel{\hat{\subseteq}} +_{1\le i\le n}\Gamma_i + (\cup_{1\le i\le n}X_i \cdot \Delta')$.

(T-BLOCK) We know that $e = \{\tau\ x = e_1;\ e_2\}$ and $\Gamma_1, x :_{X_1} \tau' \vdash e_1 : \tau''$ and $\Gamma_2, x :_{X_2} \tau', y :_Y \tau'' \vdash e_2 : \tau$ and

$$\Gamma, x :_X \tau' = (Y \cup \{\ell\}) \cdot (\Gamma_1, x :_{X_1} \tau') + (\Gamma_2, x :_{X_2} \tau')$$

where ℓ is fresh. Then, in particular, we have $\Gamma \mathrel{\hat{\subseteq}} (Y \cup \{\ell\}) \cdot \Gamma_1 + \Gamma_2$ and $X \supseteq (Y\cup\{\ell\})\triangleleft X_1 \cup X_2$. We want to prove that $\Gamma' \vdash \{\tau\ x = e_1;\ e_2\}[e'/x] : \tau$ where

$$\Gamma' = ((Y \cup \{\ell\}) \triangleleft X_1 \cup X_2) \cdot \Delta' + (Y \cup \{\ell\}) \cdot \Gamma_1 + \Gamma_2$$

then the thesis will follow by subsumption. By induction hypothesis, we get $\Gamma_1+X_1\cdot\Delta' \vdash e_1[e'/x] : \tau''$ and $(\Gamma_2, y :_Y \tau'')+X_2\cdot\Delta' \vdash e_2[e'/x] : \tau$. By definition of $+$ we have $(\Gamma_2, y :_Y \tau'')+X_2\cdot\Delta' = (\Gamma_2 + X_2 \cdot \Delta', y :_Y \tau'')^\star = \Theta, y :_{Y'} \tau''$ for some Θ and Y'. By Lemma 6.3.8 we know that Y' is the union of all coeffects in $\Gamma_2 + X_2 \cdot \Delta'$ that are not disjoint from Y.

Then, by rule (T-BLOCK), we derive $\Gamma'' \vdash \{\tau''\ y = e_1;\ e_2\}[e/x] : \tau$ with

$$\Gamma'' = (Y' \cup \{\ell\}) \cdot (\Gamma_1 + X_1 \cdot \Delta') + \Gamma_2 + X_2 \cdot \Delta'$$

Since $Y \subseteq Y'$, we have

$$\Gamma'' = Y' \cdot (\Gamma_1 + X_1 \cdot \Delta') + (Y \cup \{\ell\}) \cdot (\Gamma_1 + X_1 \cdot \Delta') + \Gamma_2 + X_2 \cdot \Delta'$$

Since the links in Δ' are fresh, we have that $X_2 \cdot \Delta' = X_2 \triangleleft \Delta'$, that is, $X_2 \cap \mathsf{links}(\Delta') \subseteq \{\mathrm{R}\}$ so $X_2 \cdot \Delta'$ is equal to Δ' except that in coeffects R is replaced with links in X_2. Moreover, by Lemma 6.3.7 $X_2 = Y$ or $X_2 \cap Y = \emptyset$. Therefore, for all $z \in \mathsf{dom}(X_2 \cdot \Delta')$, either $Y \subseteq \mathsf{coeff}(X_2 \cdot \Delta', z)$ or $Y \cap \mathsf{coeff}(X_2 \cdot \Delta', z) = \emptyset$. Applying Lemma 6.3.8 we get $\Theta = \Gamma_2 + X_2 \cdot \Delta'$ and $Y' = \mathsf{coeff}(\Theta, z)$ if $Y \subseteq \mathsf{coeff}(\Theta, z)$ for all $z \in \mathsf{dom}(\Theta)$. For all $z \in \mathsf{dom}(\Gamma_1+X_1\cdot\Delta')$ such that $\mathrm{R} \in \mathsf{coeff}(\Gamma_1+X_1\cdot\Delta', z)$, we have that $Y \subseteq \mathsf{coeff}((Y\cup\{\ell\})\cdot(\Gamma_1+X_1\cdot\Delta'), z)$. Since $(Y\cup\{\ell\})\cdot(\Gamma_1+X_1\cdot\Delta')+\Gamma_2+X_2\cdot\Delta'$

is closed and Y' is the union of all coeffects in $\Gamma_2 + X_2 \cdot \Delta'$ that are not disjoint from Y we get that $Y \subseteq \text{coeff}((Y \cup \{\ell\}) \cdot (\Gamma_1 + X_1 \cdot \Delta'), z)$ implies $Y' \subseteq \text{coeff}((Y \cup \{\ell\}) \cdot (\Gamma_1 + X_1 \cdot \Delta') + \Gamma_2 + X_2 \cdot \Delta', z)$.

Instead, if $\text{R} \notin \text{coeff}(\Gamma_1 + X_1 \cdot \Delta', z)$, then $\text{coeff}(Y' \mathbin{\hat{\triangleleft}} (\Gamma_1 + X_1 \cdot \Delta'), z) = \text{coeff}(\Gamma_1 + X_1 \cdot \Delta', z)$. Therefore

$$Y' \mathbin{\hat{\triangleleft}} (\Gamma_1 + X_1 \cdot \Delta') \mathbin{\hat{\subseteq}} (Y \cup \{\ell\}) \cdot (\Gamma_1 + X_1 \cdot \Delta') + \Gamma_2 + X_2 \cdot \Delta'$$

and so $\Gamma'' = (Y \cup \{\ell\}) \cdot (\Gamma_1 + X_1 \cdot \Delta') + \Gamma_2 + X_2 \cdot \Delta'$.

Finally

$$\begin{array}{rcl} \Gamma'' & = & (Y \cup \{\ell\}) \cdot (\Gamma_1 + X_1 \cdot \Delta') + \Gamma_2 + X_2 \cdot \Delta' \\ & = & ((Y \cup \{\ell\}) \cdot X_1) \cdot \Delta' + X_2 \cdot \Delta' + (Y \cup \{\ell\}) \cdot \Gamma_1 + \Gamma_2 \\ & = & ((Y \cup \{\ell\}) \triangleleft X_1 \cup X_2) \cdot \Delta' + (Y \cup \{\ell\}) \cdot \Gamma_1 + \Gamma_2 \end{array}$$

We want to show that $((Y \cup \{\ell\}) \triangleleft X_1 \cup X_2) \cdot \Delta' + (Y \cup \{\ell\}) \cdot \Gamma_1 + \Gamma_2 \mathbin{\hat{\subseteq}} X \cdot \Delta' + \Gamma$. Since $X \supseteq (Y \cup \{\ell\}) \triangleleft X_1 \cup X_2$ and $\Gamma \mathbin{\hat{\subseteq}} (Y \cup \{\ell\}) \cdot \Gamma_1 + \Gamma_2$ we derive $((Y \cup \{\ell\}) \triangleleft X_1 \cup X_2) \cdot \Delta' + (Y \cup \{\ell\}) \cdot \Gamma_1 + \Gamma_2 \mathbin{\hat{\subseteq}} X \cdot \Delta' + \Gamma$.

□

PROOF OF THEOREM 6.3.3

Proof: If $\Gamma' \vdash e|\mu : \tau$, we have $\Gamma' = \Gamma_1' + \Gamma_2$, $\Gamma_1' \vdash e : \tau$ and $\Gamma_2 \vdash \mu$. We know that $\Gamma_2 = \Gamma_\mu + \Theta$, where $\Gamma_\mu = x_1 :_{\{\ell_1\}} \tau_1, \ldots, x_n :_{\{\ell_n\}} \tau_n$, $\Theta = \sum_{i=1}^n \ell_i \cdot \Theta_i$, $\Theta_i \vdash \mu(x_i) : \tau_u$ and $\ell_1, \ldots, \ell_n$ are fresh links. We also know that $\Gamma_2 = \Gamma_\mu + \Theta$, where $\Gamma_\mu = x_1 :_{\{\ell_1\}} \tau_1, \ldots, x_n :_{\{\ell_n\}} \tau_n$, $\Theta = \sum_{i=1}^n \ell_i \cdot \Theta_i$, $\Theta_i \vdash \mu(x_i) : \tau_u$ and $\ell_1, \ldots, \ell_n$ are fresh links. We also know that by Lemma 6.3.5 $\Gamma_1 \vdash e : \tau$ with $\Gamma_1 \blacktriangleleft \Gamma_1'$. In the proof below we prove the theorem with as hypothesis $\Gamma \vdash e|\mu : \tau$ where $\Gamma = \Gamma_1 + \Gamma_2$ obtaining that $\Delta \vdash e'|\mu' : \tau$ and $\Gamma + \Delta \upharpoonright \Gamma = \Gamma$. We also know that $\Delta = \Gamma_3 + \Gamma_4$ and $\Gamma_3 \vdash e : \tau$ and $\Gamma_4 \vdash \mu'$. If $\Gamma_1 = \Gamma_1'$ we have also the thesis. If $\Gamma_1' = \{\ell\} \cdot \Gamma_1$ then we can apply the same non syntx directed rules applied to $\Gamma_1 \vdash e : \tau$ obtaining $\{\ell\} \triangleleft \Gamma_3 \vdash e' : \tau$ and so, since $\text{R} \notin \text{links}(\Gamma_4)$, $\{\ell\} \triangleleft \Delta \vdash e|\mu : \tau$. By Lemma 6.3.12 we have that $\{\ell\} \cdot (\Gamma + \Delta) \upharpoonright \{\ell\} \cdot \Gamma = \{\ell\} \cdot \Gamma$, that is, the thesis. The proof is by induction on the reduction relation.

(FIELD-ACCESS) We know that $x.f_k|\mu \longrightarrow v_k|\mu$, hence $e = x.f_k$, $e' = v_k$ and $\mu' = \mu$. We know that $\mu(x) = \texttt{new}\ C(v_1, \ldots, v_k, \ldots, v_m)$ with $\text{k} \in 1..m$. Since $x \in \text{dom}(\mu)$ we know exists an $h \in 1..n$ such that $x = x_h$ and so $\Theta_h \vdash \mu(x) : \tau_h$. By Lemma 6.3.6(5) we know that $\Theta_h = \sum_{i=1}^m \Theta_i'$ such that $\Theta_i' \vdash v_i : \tau_i$. By this consideration we derive that $\Theta_k' \vdash v_k : \tau_k$. We know that $\Gamma \mathbin{\hat{\subseteq}} \{\ell_h\} \cdot \Theta_h \mathbin{\hat{\subseteq}} \{\ell_h\} \triangleleft \Theta_h$. For all $y \in \text{dom}(\Theta_h)$ we know $\text{coeff}(\{\ell_h\} \triangleleft \Theta_h, y) = (\text{coeff}(\Theta_h, y) \setminus \{\text{R}\}) \cup \{\ell_h\}$ if $\text{R} \in \text{coeff}(\Theta_h, y)$, $\text{coeff}(\{\ell_h\} \triangleleft \Theta_h, y) = \text{coeff}(\Theta_h, y)$ otherwise. We know that $\{\text{R}, \ell_h\} \subseteq \text{coeff}(\Gamma, x)$, so, since Γ is closed and $\Gamma \mathbin{\hat{\subseteq}} \{\ell_h\} \triangleleft \Theta_h$, for all $y \in \text{dom}(\{\ell_h\} \triangleleft \Theta_h)$, $\ell_h \in \text{coeff}(\{\ell_h\} \triangleleft \Theta_h, y)$ implies $\text{R} \in \text{coeff}(\Gamma, y)$ and so $\text{coeff}(\Gamma, y) \supseteq (\text{coeff}(\Theta_h, y) \setminus \{\text{R}\}) \cup \{\ell_h\} \cup \{\text{R}\} \supseteq \text{coeff}(\Theta_h, y)$. By these considerations we derive that $\Gamma \mathbin{\hat{\subseteq}} \Theta_h$. We can apply rule (T-CONF) obtaining $\Gamma_2 + \Theta_k' \vdash v_k|\mu' : \tau_k$. Since $\Gamma = \Gamma + \Gamma_2 + \Theta_k'$ we have

$(\Gamma + \Gamma_2 + \Theta'_k) \restriction \Gamma = \Gamma \restriction \Gamma = \Gamma$ we have the thesis.

(FIELD-ASSIGN) We know that $x\,.\,f_k = v|\mu \longrightarrow v|\mu^{x.k=v}$, hence $e = x\,.\,f_k = v$, $e' = v$ and $\mu' = \mu^{x.k=v}$. By Lemma 6.3.6(4), we get $\Gamma_1 = \Delta_1 + \Delta_2, \Delta_1 \vdash x : C$, $\Delta_2 \vdash v : \tau$, and $\mathsf{fields}(C) = S_1\ f_1\texttt{;}, \ldots, S_m\ f_m\texttt{;}$ with $k \in 1..m$. Again by Lemma 6.3.6(1), we get $\Delta_1 = x :_{\{\mathrm{R}\}} C$. We know that $\mu(z) = \mu'(z)$ for all $z \in \mathsf{dom}(\mu) \setminus \{x\}$ and $\mu(x) = \texttt{new}\ C\,(v_1, \ldots, v_m)$ and $\mu'(x) = \texttt{new}\ C\,(v_1, \ldots, v_{k-1}, v, v_{k+1}, \ldots, v_m)$. Since $x \in \mathsf{dom}(\mu)$, there is $h \in 1..n$ such that $x = x_h$ and so $\Theta_h \vdash \mu(x) : \tau_h$ holds, with $\tau_h = C$. Applying Lemma 6.3.6(5), we derive that $\Theta_h = \Sigma_{i=1}^{m}\Theta'_i$ and $\Theta'_i \vdash v_i : S_i$, for all $i \in 1..m$. By these considerations and applying rule (T-NEW) we obtain $\Sigma_{i=1}^{k-1}\Theta'_i + \Delta_2 + \Sigma_{i=k+1}^{m}\Theta'_i \vdash \texttt{new}\ C\,(v_1, \ldots, v_{k-1}, v, v_{k+1}, \ldots, v_m) : C$.

Applying rule (T-MEM) we can type memory μ', deriving $\Gamma_\mu + (\sum_{i=1}^{h-1}\{\ell_i\} \cdot \Theta_i) + (\{\ell_h\} \cdot (\Sigma_{i=1}^{k-1}\Theta'_i + \Delta_2 + \Sigma_{i=k+1}^{m}\Theta'_i)) + (\sum_{i=h+1}^{n}\{\ell_i\} \cdot \Theta_i) \vdash \mu'$ with $\mathsf{coeff}(\Gamma_\mu, x) = \{\ell_x\}$. We know $\Gamma_\mu + (\sum_{i=1}^{h-1}\{\ell_i\} \cdot \Theta_i) + (\{\ell_h\} \cdot (\Sigma_{i=1}^{k-1}\Theta'_i + \Delta_2 + \Sigma_{i=k+1}^{m}\Theta'_i)) + (\sum_{i=h+1}^{n}\{\ell_i\} \cdot \Theta_i) = \Gamma_\mu + (\sum_{i=1}^{h-1}\{\ell_i\} \cdot \Theta_i) + \{\ell_h\} \cdot \Sigma_{i=1}^{k-1}\Theta'_i + \{\ell_h\} \triangleleft \Delta_2 + \{\ell_h\} \triangleleft \Sigma_{i=k+1}^{m}\Theta'_i + (\sum_{i=h+1}^{n}\{\ell_i\} \cdot \Theta_i)$. We have two cases for all $y \in \mathsf{dom}(\Delta_2)$:

- $\mathrm{R} \in \mathsf{coeff}(\Delta_2, y)$
  We have $\mathsf{coeff}(\{\ell_h\} \triangleleft \Delta_2, y) = (\mathsf{coeff}(\Delta_2, y) \setminus \{\mathrm{R}\}) \cup \{\ell_h\}$. We have $\{\mathrm{R}, \ell_h\} \subseteq \mathsf{coeff}(\Gamma, x)$, and, since $\Gamma \mathrel{\hat{\subseteq}} \Delta_2$ we know $\mathsf{coeff}(\Gamma, y) \supseteq \mathsf{coeff}(\Delta_2, y)$ and in particular $\mathrm{R} \in \mathsf{coeff}(\Gamma, y)$. By the fact that Γ is closed we know that $\ell_h \in \mathsf{coeff}(\Gamma, y)$. We can also conclude that $\mathsf{coeff}(\Gamma, y) \supseteq \mathsf{coeff}(\Delta_2, y) \cup \{\ell_h\} \supseteq (\mathsf{coeff}(\Delta_2, y) \setminus \{\mathrm{R}\}) \cup \{\ell_h\}$.
- $\mathrm{R} \notin \mathsf{coeff}(\Delta_2, y)$
  We have $\mathsf{coeff}(\{\ell_h\} \triangleleft \Delta_2, y) = \mathsf{coeff}(\Delta_2, y)$. Since $\Gamma \mathrel{\hat{\subseteq}} \Delta_2$ we derive $\mathsf{coeff}(\Gamma, y) \supseteq \mathsf{coeff}(\Delta_2, y)$.

By these considerations we have that $\{\ell_h\} \triangleleft \Delta_2 \mathrel{\hat{\subseteq}} \Gamma$, so $\{\ell_h\} \triangleleft \Delta_2 \mathrel{\hat{\subseteq}} \Gamma$.

Since $\Gamma_\mu \mathrel{\hat{\subseteq}} \Gamma$, $(\sum_{i=1}^{h-1}\{\ell_i\} \cdot \Theta_i) \mathrel{\hat{\subseteq}} \Gamma$, $\{\ell_h\} \cdot \Sigma_{i=1}^{k-1}\Theta'_i \mathrel{\hat{\subseteq}} \Gamma$, $\{\ell_h\} \triangleleft \Delta_2 \mathrel{\hat{\subseteq}} \Gamma$,$\{\ell_h\} \triangleleft \Sigma_{i=k+1}^{m}\Theta'_i \mathrel{\hat{\subseteq}} \Gamma$, $(\sum_{i=h+1}^{n}\{\ell_i\} \cdot \Theta_i) \mathrel{\hat{\subseteq}} \Gamma$. We derive $\Gamma_\mu + (\sum_{i=1}^{h-1}\{\ell_i\} \cdot \Theta_i) + \{\ell_h\} \cdot \Sigma_{i=1}^{k-1}\Theta'_i + \{\ell_h\} \triangleleft \Delta_2 + \{\ell_h\} \triangleleft \Sigma_{i=k+1}^{m}\Theta'_i + (\sum_{i=h+1}^{n}\{\ell_i\} \cdot \Theta_i) \mathrel{\hat{\subseteq}} \Gamma$. Applying rule (T-CONF), we obtain $\Delta = \Gamma_\mu + (\sum_{i=1}^{h-1}\{\ell_i\} \cdot \Theta_i) + \{\ell_h\} \cdot \Sigma_{i=1}^{k-1}\Theta'_i + \{\ell_h\} \triangleleft \Delta_2 + \{\ell_h\} \triangleleft \Sigma_{i=k+1}^{m}\Theta'_i + (\sum_{i=h+1}^{n}\{\ell_i\} \cdot \Theta_i) + \Delta_2$. We know that $\Gamma + \Delta = \Gamma$. By the fact that $\Delta + \Gamma \restriction \Gamma = \Gamma \restriction \Gamma = \Gamma$ we obtain the thesis.

(NEW) We know that $\texttt{new}\ C(v_1, \ldots, v_n)|\mu \longrightarrow x|\mu, x \mapsto \texttt{new}\ C(v_1, \ldots, v_n)$ with $x \notin \mathsf{dom}(\mu)$ hence $e = \texttt{new}\ C(v_1, \ldots, v_n)$, $e' = x$ and $\mu' = \mu, x \mapsto \texttt{new}\ C(v_1, \ldots, v_n)$. Since we have $\mu(y) = \mu'(y)$ for all $y \in \mathsf{dom}(\mu)$ and $\mathsf{dom}(\mu') = \mathsf{dom}(\mu) \cup \{x\}$ and $\mu'(x) = \texttt{new}\ C(v_1, \ldots, v_n)$ we can apply rule (T-MEM) deriving $\Gamma_\mu + x :_{\ell_{m+1} C} + \Theta + (\ell_{m+1} \cdot \Gamma_1) \vdash \mu'$, where $\Gamma_1 \vdash \mu'(x) : C$, ℓ_{m+1} fresh and $\ell_{m+1} \notin \mathsf{links}(\Gamma)$. By rule (T-VAR) we know $x :_{\{\mathrm{R}\}} C \vdash x : C$. By rule (T-CONF) we derive $\Delta = \Gamma_\mu + x :_C \{\ell_{m+1}\} + \Theta + (\{\ell_{m+1}\} \cdot \Gamma_1) + x :_{\{\mathrm{R}\}} C$ and $\Gamma_\mu + x :_C \{\ell_{m+1}\} + \Theta + (\{\ell_{m+1}\} \cdot \Gamma_1) + x :_{\{\mathrm{R}\}} C \vdash x|\mu' : C$. We know $\Gamma + \Delta = \Gamma_\mu + x :_{\{\ell_{m+1}\}} C + \Theta + (\{\ell_{m+1}\} \cdot \Gamma_1) + x :_{\{\mathrm{R}\}} C + \Gamma =$

$\Gamma + x :_{\{\mathrm{R},\ell_{m+1}\}} C + (\{\ell_{m+1}\} \cdot \Gamma_1)$. We know that $\ell_{m+1} \notin \mathsf{coeff}(\Gamma, y)$ for all $y \in \mathsf{dom}(\Gamma)$ and we know that, since Γ is closed, $\mathrm{R} \in \mathsf{coeff}(\Gamma, y)$ and $\mathrm{R} \in \mathsf{coeff}(\Gamma, z)$ implies $\mathsf{coeff}(\Gamma, y) = \mathsf{coeff}(\Gamma, z)$ for all $y, z \in \mathsf{dom}(\Gamma)$. We also know that $x \notin \mathsf{dom}(\Gamma)$, so $\Gamma + x :_{\{\ell_{m+1},\mathrm{R}\}} C = (\Gamma, x :_{\{\ell_{m+1},\mathrm{R}\}} C)^\star = \Gamma', x :_X C$. By Lemma 6.3.8 and by the observations above we have 2 cases for all $y \in \mathsf{dom}(\Gamma)$:

- $\mathrm{R} \notin \mathsf{coeff}(\Gamma, y)$ implies $\mathsf{coeff}(\Gamma', y) = \mathsf{coeff}(\Gamma, y)$
- $\mathrm{R} \in \mathsf{coeff}(\Gamma, y)$ implies $\mathsf{coeff}(\Gamma', y) = \mathsf{coeff}(\Gamma, y) \cup \{\ell_{m+1}, \mathrm{R}\} = \mathsf{coeff}(\Gamma, y) \cup \{\ell_{m+1}\}$

We know $\Gamma_1 \hat{\subseteq} \Gamma \hat{\subseteq} \Gamma'$ and $(\Gamma', x :_X C) + (\{\ell_{m+1}\} \cdot \Gamma_1) = (\Gamma', x :_X C) + (\{\ell_{m+1}\} \triangleleft \Gamma_1)$.

If $\mathrm{R} \in \mathsf{coeff}(\Gamma_1, y)$ then $\mathsf{coeff}(\{\ell_{m+1}\} \triangleleft \Gamma_1, y) = (\mathsf{coeff}(\Gamma_1, y) \setminus \{\mathrm{R}\}) \cup \{\ell_{m+1}\}$ and, since $\mathrm{R} \in \mathsf{coeff}(\Gamma, y)$, $\mathsf{coeff}(\Gamma', y) = \mathsf{coeff}(\Gamma, y) \cup \{\ell_{m+1}\} \supseteq \mathsf{coeff}(\Gamma_1, y) \cup \{\ell_{m+1}\}$ for all $y \in \mathsf{dom}(\Gamma_1)$. if $\mathrm{R} \notin \mathsf{coeff}(\Gamma_1, y)$ then $\mathsf{coeff}(\{\ell_{m+1}\} \triangleleft \Gamma_1, y) = \mathsf{coeff}(\Gamma_1, y) \subseteq \mathsf{coeff}(\Gamma', y)$ for all $y \in \mathsf{dom}(\Gamma_1)$. By these observations we can conclude that $(\Gamma', x :_X C) + (\{\ell_{m+1} \triangleleft \Gamma_1\}) = \Gamma', x :_X C = \Gamma + \Delta$. By the fact that $\ell_{m+1} \notin \mathsf{links}(\Gamma)$ and $\mathsf{coeff}(\Gamma + \Delta, z) = \mathsf{coeff}(\Gamma, z)$ or $\mathsf{coeff}(\Gamma + \Delta, z) = \mathsf{coeff}(\Gamma, z) \cup \{\ell_{m+1}\}$ for all $z \in \mathsf{dom}(\Gamma)$ we derive that $(\Gamma + \Delta) \restriction \Gamma = \Gamma$, that is, the thesis.

(CTX) We have $e = \mathcal{E}[e_1]$ and $e' = \mathcal{E}[e_1']$ and $e_1|\mu \to e_1'|\mu'$. By Lemma 6.3.9, $\Gamma_1 = \Delta_1 + X \cdot \Delta_2$, $\Delta_1 + x :_X \tau' \vdash \mathcal{E}[x] : \tau$ and $\Delta_2 \vdash e_1 : \tau'$ and we can impose that if $\ell \in \mathsf{coeff}(\Delta_1, x)$ and $\ell \in X$ then $\mathsf{coeff}(\Delta_1, x) = X$. We get $\Gamma = \Gamma_1 + \Gamma_2 = \Delta_1 + X \cdot \Delta_2 + \Gamma_2$ and, since $\mathrm{R} \notin \mathsf{coeff}(\Gamma_2, y)$ for every $y \in \mathsf{dom}(\Gamma_2)$ by rule (T-MEM) we have $\Gamma_2 = X \cdot \Gamma_2$, hence $\Gamma = \Delta_1 + X \cdot (\Delta_2 + \Gamma_2)$. We set $\Gamma' = \Delta_2 + \Gamma_2$. By induction hypothesis, we get $\Delta' \vdash e_1'|\mu' : \tau'$ with $\Delta' + \Gamma' \restriction \Gamma' = \Gamma'$. By rule (T-CONF), we know that $\Delta' = \Delta_1' + \Delta_2'$ with $\Delta_1' \vdash e_1' : \tau'$ and $\Delta_2' \vdash \mu'$. By Lemma 6.3.10, we get $\Delta_1 + X \cdot \Delta_1' \vdash \mathcal{E}[e_1'] : \tau$ and so $\Delta_1 + X \cdot \Delta_1' + \Delta_2' \vdash \mathcal{E}[e_1']|\mu' : \tau$. We have $\Delta = \Delta_1 + X \cdot \Delta_1' + \Delta_2' = \Delta_1 + X \cdot \Delta_1' + X \cdot \Delta_2' = \Delta_1 + X \cdot \Delta'$. We get the thesis, that is, $\Delta + \Gamma \restriction \Gamma = \Gamma$, by Lemma 6.3.12, since $\mathsf{links}(\Delta_1) \cap (\mathsf{links}(X \triangleleft (\Delta' + \Gamma')) \cup \mathsf{links}(X \triangleleft \Gamma')) = \{\mathrm{R}\}$ or $\mathsf{links}(\Delta_1) \cap (\mathsf{links}(X \triangleleft (\Delta' + \Gamma')) \cup \mathsf{links}(X \triangleleft \Gamma')) = \{\mathrm{R}\} \cup X$ if exists $x \in \mathsf{dom}(\Delta)$ such that $\mathsf{coeff}(\Delta, x) = X$ and $\mathsf{dom}(\Delta_1) \subseteq \mathsf{dom}(\Gamma_{\mathrm{R}}) \subseteq \mathsf{dom}(X \cdot \Gamma')$.

(BLOCK) $\{\tau\ x = v;\ e\}|\mu \to e[v/x]|\mu$ Applying Lemma 6.3.6(7) we obtain $\Gamma', x :_X \tau \vdash e : \tau'$ and $\Sigma \vdash v : \tau$ such that $\Gamma_1 = (X \cup \{\ell\}) \cdot \Sigma + \Gamma'$. If v is a reference then by Lemma 6.3.6(1) we know $\Sigma = x :_{\{\mathrm{R}\}} \tau$ otherwise if v is a primitive value then we know by (T-PRIMITIVE) $\Sigma = \emptyset$. Applying Lemma 6.3.13 we obtain $\Gamma'' \vdash e[v/x] : \tau'$ and $\Gamma'' \hat{\subseteq} (X \cup \{\ell\}) \cdot \Sigma + \Gamma'$. Knowing that $\Gamma'' + \Gamma_2 + \Gamma = \Delta + \Gamma = \Gamma$ we obtain $\Gamma \restriction \Gamma = \Gamma$, that is, the thesis.

□

COROLLARY 6.3.14 : If $\Gamma \vdash e|\mu : \tau$ and $\langle e, \mu\rangle \to^\star \langle e', \mu'\rangle$, then $\Delta \vdash e'|\mu' : \tau$ for some Δ such that $(\Gamma + \Delta) \restriction \Gamma = \Gamma$.

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

Thanks to the fact that reduction preserves (initial) sharing, we can *statically detect* lent references (Definition 6.2.3) and capsule expressions (Definition 6.2.4) just looking at coeffects, as stated below.

THEOREM 6.3.15 (Lent reference): If $\Gamma \vdash e|\mu : C$, $x \in \mathsf{dom}(\Gamma)$ with $\mathrm{R} \notin \mathsf{coeff}(\Gamma, x)$, and $e|\mu \to^\star y|\mu'$, then $x \bowtie_{\mu'} y$ does not hold.

Proof: By Theorem 6.3.3 we have $\Delta \vdash y|\mu' : C$, for some Δ such that $(\Gamma + \Delta)\restriction\Gamma = \Gamma$. By inversion, we have $y :_{\{\mathrm{R}\}} C \vdash y : C$ with $\Delta = \Delta' + y :_{\{\mathrm{R}\}} C$, hence $\mathrm{R} \in \mathsf{coeff}(\Delta, y)$. Assume $x \bowtie_{\mu'} y$. By Lemma 6.3.2, we have $\mathsf{coeff}(\Delta, x) = \mathsf{coeff}(\Delta, y)$, thus $\mathrm{R} \in \mathsf{coeff}(\Delta, x)$. Since $(\Gamma + \Delta)\restriction\Gamma = \Gamma$, $\mathrm{R} \in \mathsf{coeff}(\Gamma + \Delta, x)$ and $x \in \mathsf{dom}(\Gamma)$, we also have $\mathrm{R} \in \mathsf{coeff}(\Gamma, x)$, contradicting the hypothesis. □

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

COROLLARY 6.3.16 (Capsule expression): If $\Gamma \vdash e|\mu : C$, with $\mathsf{capsule}(\Gamma)$, and $e|\mu \to^\star y|\mu'$, then, for all $x \in \mathsf{dom}(\Gamma)$, $x \bowtie_{\mu'} y$ does not hold.

Proof: Let $x \in \mathsf{dom}(\Gamma)$.

The hypothesis $\mathsf{capsule}(\Gamma)$ means that each variable in Γ is lent, in the sense of Theorem 6.3.15, that is, $\mathrm{R} \notin \mathsf{coeff}(\Gamma, x)$. Then, by Theorem 6.3.15, $x \bowtie_{\mu'} y$ does not hold.

□

Note that, in particular, Corollary 6.3.16 ensures that no free variable of e can access the reachable object graph of the final result y. Notice also that assuming $\mathsf{capsule}(\Gamma)$ is the same as assuming $\mathsf{capsule}(\Delta)$ where $\Gamma = \Delta + \Delta'$ and Δ is the context that types the expression e, because no R link can occur in the context that types the memory.

6.4 Case study: type modifiers for uniqueness and immutability

The coeffect system in the previous section tracks sharing among variables possibly introduced by reduction. In this section, we check the effectiveness of the approach to model specific language features related to sharing and mutation, taking as challenging case study those proposed by Giannini et al. [32, 33], whose common key ideas are the following:

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

For instance, consider the following version of Example 6.2.5 decorated with modifiers:

EXAMPLE 6.4.1 :

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

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

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

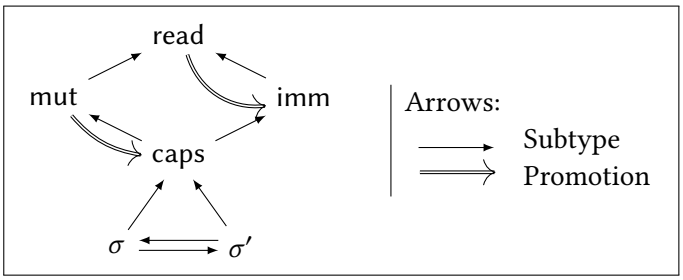

FIGURE 6.5 Type modifiers and their relationships

guarantee that the portion of memory denoted by `mycaps` cannot be modified through another reference. That is, the immutability property is detected as a conjunction of the read-only restriction and the capsule property.

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

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

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

We formalize the features illustrated above by building, on top of that of the previous section, a type-and-coeffect system whose key advantage is that detection of caps and imm types is *straightforward* from the coeffects, through a simple *promotion*[4] rule, since they exactly express the desired properties.

Type-and-coeffect contexts are, as before, of shape $x_1 :_{X_1} \tau_1, \ldots, x_n :_{X_n} \tau_n$, where types are either primitive types or of shape C^{M}, with M modifier. We assume that fields can be declared either imm or mut, whereas the modifiers caps and read are only used for local variables. Besides those, which are written by the programmer in source code, modifiers include a numerable set of *seals* σ which are only internally used by the type system, as will be explained later.

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

- The partial order must take into account subtyping as well, defined by
  - $\tau \leq \tau'$ if either $\tau = \tau'$ primitive type, or $\tau = C^{\mathrm{M}}$, $\tau' = C^{\mathrm{M}'}$, with
  - $\mathrm{M} \leq \mathrm{M}'$ induced by $\sigma \leq \sigma'$, $\sigma \leq$ caps, caps $\leq$ mut, caps $\leq$ imm, mut $\leq$ read, imm $\leq$ read, see Figure 6.5.
- In the sum of two contexts, denoted $\Gamma \oplus \Delta$, variables of a caps or σ type cannot occur in both; that is, they are handled *linearly*.

Combination of modifiers, denoted $\mathrm{M}[\mathrm{M}']$, is defined by

4 This terminology is chosen to emphasize the analogy with promotion in the sense of linear logic and graded type systems.

$$\mathrm{M}[\mathrm{M}'] = \mathrm{M} \text{ if } \mathrm{M} \leq \mathsf{imm} \qquad \mathsf{mut}[\mathrm{M}] = \mathrm{M}$$

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

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

The typing rules are given in Figure 6.6. We only comment on the novelties with respect to Section 6.3.

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

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

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

Rule (T-IMM) generalizes rule (T-PRIM) of the previous type system, allowing the links with the result to be removed, to immutable types. For instance, assuming the following variant of Example 6.2.2 (recall that the default modifier mut can be omitted):

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

the following derivable judgment

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

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

The new rule (T-PROM) plays a key role, since, as already mentioned, it detects that an expression is a capsule thanks to its coeffects, and *promotes* its type accordingly. The basic idea is that a mut (resp. read) expression can be promoted to caps (resp. imm) provided that there are no free variables connected to the result with modifier read or mut. However, to guarantee that subject holds, the same promotion should be possible for runtime expressions, which may contain free variables which actually are mut references generated during reduction. To this end, the rule allows mut variables connected to the result[5]. Such variables become *sealed* as an effect of the promotion, leading

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

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

$$\text{(T-SUB)}\ \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau}\ \tau' \leq \tau \qquad \text{(T-VAR)}\ \frac{}{\emptyset \cdot \Gamma \oplus x :_{\{\mathrm{R}\}} \tau \vdash x : \tau}$$

$$\text{(T-CONST)}\ \frac{}{\emptyset \triangleleft \Gamma \vdash k : P_k}$$

$$\text{(T-FIELD-ACCESS)}\ \frac{\Gamma \vdash e : C^{\mathrm{M}}}{\Gamma \vdash e.f_i : \tau_i[\mathrm{M}]}\ \begin{array}{l}\mathsf{fields}(C) = \tau_1\, f_1; \ldots \tau_n\, f_n; \\ i \in 1..n\end{array}$$

$$\text{(T-FIELD-ASSIGN)}\ \frac{\Gamma \vdash e : C^{\mathsf{mut}} \quad \Delta \vdash e' : \tau_i}{\Gamma \oplus \Delta \vdash e.f_i = e' : \tau_i}\ \begin{array}{l}\mathsf{fields}(C) = \tau_1\, f_1; \ldots \tau_n\, f_n; \\ i \in 1..n\end{array}$$

$$\text{(T-NEW)}\ \frac{\Gamma_i \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Gamma_1 \oplus \ldots \oplus \Gamma_n \vdash \mathtt{new}\ C(e_1, \ldots, e_n) : C^{\mathsf{mut}}}\ \mathsf{fields}(C) = \tau_1\, f_1; \ldots \tau_n\, f_n;$$

$$\text{(T-INVK)}\ \frac{\Gamma_0 \vdash e_0 : C^{\mathrm{M}} \quad \Gamma_i \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Gamma_0' \ldots \Gamma_n' \vdash e_0.m(e_1, \ldots, e_n) : \tau}\ \begin{array}{l}\mathsf{mtype}(C, m) \equiv^{\mathsf{fr}} \mathrm{M}^{X_0}, \tau_1^{X_1} \ldots \tau_n^{X_n} \to \tau \\ \Gamma_i' = X_i \cup \{\ell_i\} \cdot \Gamma_i \\ \ell_0, \ldots, \ell_n \text{ fresh}\end{array}$$

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

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

$$\text{(T-REF)}\ \frac{}{x :_{C^{\mathrm{M}}} \{\mathrm{R}\} \Vdash x : C^{\mathrm{M}}}\ \mathrm{M}{=}\mathsf{mut} \text{ or } \mathrm{M}{=}\sigma$$

$$\text{(T-IMM-REF)}\ \frac{}{x :_{\{\ell\}} C^{\mathsf{imm}} \Vdash x : C^{\mathsf{imm}}}\ \ell \text{ fresh} \qquad \text{(T-MEM-CONST)}\ \frac{}{\emptyset \Vdash k : P_k}$$

$$\text{(T-OBJ)}\ \frac{\Gamma_i \Vdash v_i : \tau_i[\mathrm{M}] \quad \forall i \in 1..n}{\Gamma_1 + \cdots + \Gamma_n \Vdash [v_1, \ldots, v_n]^C : C^{\mathrm{M}}}\ \mathsf{fields}(C) = \tau_1\, f_1; \ldots \tau_n\, f_n;$$

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

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

FIGURE 6.6 Adding modifiers and immutability

to the context $\Gamma[\sigma]$, obtained from Γ by combining modifiers of variables connected to the result with σ. Formally, if $\Gamma = x_1 :_{X_1} \tau_1, \ldots, x_n :_{X_n} \tau_n$,

$\Gamma[\sigma] = x_1 :_{X_1} \tau'_1, \ldots, x_n :_{X_n} \tau'_n$ where $\tau'_i = \tau_i[\sigma]$ if $\mathrm{R} \in X_i$, $\tau'_i = \tau_i$ otherwise

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

This highlights once again the analogy with the promotion rule for the (graded) bang modality of linear logic [15], where, in order to introduce a modality on the right-hand side of a sequent, one has to modify the left-hand side accordingly.[6] We detail in the following how sealed variables are internally used by the type system to guarantee subject reduction.

Rule (T-MEM) is also analogous to that in Figure 6.3. However, typechecking objects is modeled by an ad-hoc judgment $\Vdash$, where references can only be mut, imm, or σ (read and caps are source-only notions), and subsumption is not included. As a consequence, rule (T-OBJ) imposes that a reference reachable from an imm reference or field should be tagged imm as well, and analogously for seals.

Rule (T-CONF) is as in Figure 6.3. Note that we use the sum of contexts + from the previous type system, since the linear treatment of caps and σ variables is only required in source code.

As in the previous type system, the rules in Figure 6.6 lead to an algorithm which inductively computes the coeffects of an expression. The only relevant novelty is rule (T-PROM), assumed to be applied *only when needed*, that is, when we typecheck the initialization expression of a local variable declared caps, or the argument of a method call where the corresponding parameter is declared caps. Rule (T-IMM) is applied, as (T-PRIM) before, only once, whenever an expression has either a primitive or an immutable type. Subsumption rule (T-SUB) only handles types, and can be replaced by a more verbose version of the rules with subtyping conditions where needed. In the other cases, rules are syntax-directed, that is, the coeffects of the expression in the consequence are computed as a linear combination of those of the subexpressions, where the basis is the rule for variables.

We illustrate now the use of seals to preserve types during reduction. For instance, consider again Example 6.2.2:

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

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

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

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

6 This is just an analogy, making it precise is an interesting direction for future work.

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

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

$$\begin{aligned} e_0 \mid \mu_0 &\longrightarrow e_1 \mid \mu_1 = \{\texttt{B z = w; x.f1=y;new C(z,z)}\} \mid \mu \cup \{\texttt{w} \mapsto [2]^{\texttt{B}}\} \\ &\longrightarrow e_2 \mid \mu_1 = \texttt{x.f1=y;new C(w,w)} \mid \mu \cup \{\texttt{w} \mapsto [2]^{\texttt{B}}\} \end{aligned}$$

Whereas sharing preservation, in the sense of Theorem 6.3.3, clearly still holds, to preserve the caps type of the initial expression the (T-PROM) promotion rule should be applicable to e_1 and e_2 as well. However, in the next steps w is a free variable connected to the result; for instance for e_1 we derive:

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

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

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

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

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

THEOREM 6.4.2 (Subject Reduction): If $\Gamma \vdash e|\mu : \tau$ and $e|\mu \longrightarrow e'|\mu'$ then

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

To prove this theorem we need some auxiliary definitions and lemmas.

DEFINITION 6.4.3 : $\Gamma \blacktriangleleft \Gamma'$ if

1. $\Gamma' = \Gamma$ or
2. $\Gamma' = \Gamma[\sigma]$ or
3. $\Gamma' = \{\ell\} \cdot \Gamma$ with ℓ fresh, or
4. $\Gamma' = \{\ell\} \cdot \Gamma[\sigma]$ with ℓ fresh.

LEMMA 6.4.4 : If $\mathcal{D} : \Gamma \vdash e : \tau$, then there is a subderivation $\mathcal{D}' : \Gamma' \vdash e : \tau'$ of $\mathcal{D}$ ending with a syntax-directed rule and $\Gamma' \blacktriangleleft \Gamma$.

LEMMA 6.4.5 : If, for all $x \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta)$, $\mathsf{modif}(\Gamma, x) \leq \mathsf{modif}(\Delta, x)$, then, for all $x \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta)$, $\mathsf{modif}(\Gamma[\sigma], x) \leq \mathsf{modif}(\Delta[\sigma], x)$.

Proof: We consider only contexts for expressions, so modifiers can be only imm, mut and σ. We also consider $x :_X \tau^{\mathrm{M}} \in \Gamma$ and $x :_{X'} \tau^{\mathrm{M}'} \in \Delta$. We have three cases:

- $\mathrm{M} = \mathrm{M}'$
  Immediate
- $\mathrm{M} = \sigma$ and $\mathrm{M}' = \mathsf{imm}$
  We know $\sigma[\sigma'] = \sigma$ and $\mathsf{imm}[\sigma'] = \mathsf{imm}$, so we have the thesis
- $\mathrm{M} = \sigma$ and $\mathrm{M}' = \mathsf{mut}$
  We know $\sigma[\sigma'] = \sigma$ and $\mathsf{mut}[\sigma'] = \sigma'$, so we have the thesis

□

LEMMA 6.4.6 : Let $\Gamma, \Delta \vdash \mu$ where, $\Delta = x_1 :_X C_1^{\sigma}, \ldots, x_n :_X C_n^{\sigma}$ and, for all $x \in \mathsf{dom}(\Gamma)$, $\mathsf{coeff}(\Gamma, x) \cap X = \emptyset$. Let $\mathrm{M} \notin \{\mathsf{read}, \mathsf{caps}\}$ be a modifier and $\Theta = x_1 :_{X_1} C_1^{\mathrm{M}}, \ldots, x_n :_{X_n} C_n^{\mathrm{M}}$ be such that

- $\mathrm{M} = \mathsf{imm}$ implies $X_i = \{\ell_i\}$, with ℓ_i fresh for all $i \in 1..n$.
- $\mathrm{M} \neq \mathsf{imm}$ implies $X_i = X$ for all $i \in 1..n$.

Then, $\Gamma, \Theta \vdash \mu$ holds.

LEMMA 6.4.7 : Let $\Gamma + \Gamma_\mu \vdash \mu$, where

- $\Gamma_\mu = \sum_{z \in \mathsf{dom}(\mu)} z :_{\{\ell_z\}} C_z^{\mathrm{M}_z}$ with ℓ_z fresh for all $z \in \mathsf{dom}(\mu)$;
- $\Gamma = \sum_{z \in \mathsf{dom}(\mu)} \{\ell_z\} \triangleleft \Gamma_z$ where $\Gamma_z \Vdash \mu(z) : C_z^{\mathrm{M}_z}$ for all $z \in \mathsf{dom}(\mu)$;
- $x, y \in \mathsf{dom}(\mu)$ and $\mu(x) = [v_1, \ldots, v_m]^{C_x}$, $\mathsf{fields}(C_x) = \tau_1\ f_1\mathbf{;}\ \ldots \tau_m\ f_m\mathbf{;}$, $\Gamma_x = \sum_{j=1}^{m} \Gamma'_j$, $\Gamma'_j \Vdash v_j : \tau_j[\mathrm{M}_x]$, for all $j \in 1...m$, and $\tau_i[\mathrm{M}_x] = C_y^{\mathrm{M}_y}$ and $\Gamma'_i = y' :_Y C_y^{\mathrm{M}_y}$ for some $i \in 1...m$.

Let Δ be such that $\Delta = \{\ell_x\} \triangleleft \Delta_x + \sum_{z \in \mathsf{dom}(\mu) \setminus \{x\}} \{\ell_z\} \triangleleft \Gamma_z$ with $\Delta_x = y :_Y C_y^{\mathrm{M}_y} + \sum_{j=1}^{i-1} \Gamma'_j + \sum_{j=i+1}^{m} \Gamma'_j$. Then, $\Delta + \Gamma_\mu \vdash \mu^{x.i=y}$ holds and $\mathsf{modif}(\Gamma + \Gamma_\mu, z) = \mathsf{modif}(\Delta + \Gamma_\mu, z)$ for all $z \in \mathsf{dom}(\mu)$

PROOF OF THEOREM 6.4.2

Proof: Since the proof for condition $(\Gamma' + \Delta') \restriction \Gamma' = \Gamma'$, for $\Gamma' = \mathsf{erase}(\Gamma)$ and $\Delta' = \mathsf{erase}(\Delta)$ is analogous to the proof for Theorem 6.3.3 in this proof we focus on the condition for all $x \in \mathsf{dom}(\Gamma)$, $\mathsf{modif}(\Gamma, x) \leq \mathsf{modif}(\Delta, x)$. If $\Gamma \vdash e|\mu : \tau$, we have $\Gamma = \Gamma_1 + \Gamma_2$, $\Gamma_1 \vdash e : \tau$ and $\Gamma_2 \vdash \mu$. We also know that $\Gamma_2 = \Gamma_\mu + \Theta$, where $\Gamma_\mu = x_1 :_{\{\ell_1\}} \tau_1, \ldots, x_n :_{\{\ell_n\}} \tau_n$, $\Theta = \sum_{i=1}^{n} \ell_i \cdot \Theta_i$, $\Theta_i \vdash \mu(x_i) : \tau_u$ and $\ell_1, \ldots, \ell_n$ are fresh links. The proof is by induction on the reduction relation.

(FIELD-ASSIGN) By Lemma 6.4.4 and rule (T-ASSIGN) we have $\Gamma_1'' \vdash x : \tau_i$ and $\Gamma_2'' \vdash v : \tau_i$ such that $\Gamma_1'' + \Gamma_2'' \blacktriangleleft \Gamma_1$ with $\mathsf{fields}(C) = \tau_1\, f_1\,;\, \ldots \tau_n\, f_n\,;$ and $i \in 1..n$. We have two interesting cases considering $v = y$:

- $\tau_i = D^{\mathsf{imm}}$
  Since $\Gamma_2'' \vdash y : D^{\mathsf{imm}}$ we know that $\mathsf{modif}(\Gamma_2'', y) \neq \mathsf{mut}$ and so $\mathsf{modif}(\Gamma, y) \neq \mathsf{mut}$. By rule (T-VAR) we have $y :_Y D^{\mathsf{imm}} \vdash y : D^{\mathsf{imm}}$. By applying the same non syntax directed rules applied to $\Gamma_1'' + \Gamma_2'' \vdash x\,.\,f_i = y : D^{\mathsf{imm}}$ we have $y :_Y D^{\mathsf{imm}} \vdash y : \tau$. We have two cases:
  - $\mathsf{modif}(\Gamma, y) = \mathsf{imm}$
    By Lemma 6.4.7 and applying rule (T-CONF) we have the thesis
  - $\mathsf{modif}(\Gamma, y) = \sigma$
    By Lemma 6.4.6 applied to variables with coeffect $\mathsf{coeff}(\Gamma_2, y)$ and with $\mathrm{M} = \mathsf{imm}$, by Lemma 6.4.7 and by applying rule (T-CONF) we have the thesis
- $\tau_i = D^{\mathsf{mut}}$
  Since $\Gamma_1'' \vdash x : C^{\mathsf{mut}}$ and $\Gamma_2'' \vdash y : D^{\mathsf{mut}}$, we have $\mathsf{modif}(\Gamma_1'', x) \neq \mathsf{mut}$ and $\mathsf{modif}(\Gamma_2'', y) \neq \mathsf{imm}$, so $\mathsf{modif}(\Gamma, x) \neq \mathsf{imm}$ and $\mathsf{modif}(\Gamma, y) \neq \mathsf{imm}$. We have four cases:
  - $\mathsf{modif}(\Gamma, x) = \mathsf{mut}$ and $\mathsf{modif}(\Gamma, y) = \mathsf{mut}$
    We can apply the same non syntax directed rules applied to $\Gamma_1'' + \Gamma_2'' \vdash x\,.\,f_i = y : D^{\mathsf{imm}}$ to $\Gamma_2'' \vdash v : \tau_i$. By Lemma 6.4.7 and applying rule (T-CONF) we have the thesis
  - $\mathsf{modif}(\Gamma, x) = \mathsf{mut}$ and $\mathsf{modif}(\Gamma, y) = \sigma$
    We know by Lemma 6.4.4 and rule (T-VAR) that $x :_{X''} \tau'' \vdash x : \tau''$. We can apply rule (T-VAR) and the same non syntax-directed rules applied to x on y to obtain $y :_{X'} D^{\mathsf{modif}(\Gamma_1'' + \Gamma_2'', x)} \vdash y : D^{\mathsf{mut}}$. If we apply the same non syntax-directed rules applied to $\Gamma_1'' + \Gamma_2'' \vdash x\,.\,f_i = y : \tau_i$ we have $y :_X D^{\mathsf{mut}} \vdash y : \tau$. By Lemma 6.4.6 applied to variables with coeffect $\mathsf{coeff}(\Gamma_2, y)$ and with $\mathrm{M} = \mathsf{mut}$, by Lemma 6.4.7 and applying rule (T-CONF) we have the thesis.
  - $\mathsf{modif}(\Gamma, x) = \sigma$ and $\mathsf{modif}(\Gamma, y) = \sigma'$
    Reasoning similar to that above. We also apply Lemma 6.4.6 with $\mathrm{M} = \sigma$.
  - $\mathsf{modif}(\Gamma, x) = \sigma$ and $\mathsf{modif}(\Gamma, y) = \mathsf{mut}$
    We can apply the same non syntax-directed rules applied to $\Gamma_1'' + \Gamma_2'' \vdash x\,.\,f_i = y : \tau_i$ to $\Gamma_2'' \vdash y : \tau_i$ to have $y :_Y D^{\mathsf{mut}} \vdash y : \tau$. By Lemma 6.4.6 applied to variables with coeffect $\mathsf{coeff}(\Gamma_2, x)$ and with $\mathrm{M} = \mathsf{mut}$, by Lemma 6.4.7 and applying rule (T-CONF) we have the thesis.

(BLOCK) We have $\{\tau'\ x = v\,;\ e\}$. By Lemma 6.4.4 and rule (T-BLOCK) we have a contexts $(X \cup \{\ell\}) \cdot \Gamma_1'' + \Gamma_2'' \blacktriangleleft \Gamma_1$ such that $(X \cup \{\ell\}) \cdot \Gamma_1'' + \Gamma_2'' \vdash \{\tau'\ x = v\,;\ e\} : \tau$, $\Gamma_1'' \vdash v : \tau'$ and $\Gamma_2'', x :_X \tau' \vdash e : \tau$. By Lemma 6.3.13 we

have that $\Delta'' \vdash e'[v/x] : \tau$ with $\Delta'' \hat{\subseteq} X \cdot \Gamma_1' + \Gamma_2'' \hat{\subseteq} (X \cup \{\ell\}) \cdot \Gamma_1'' + \Gamma_2''$. By applying the same non syntax directed rules applied to $(X \cup \{\ell\}) \cdot \Gamma_1'' + \Gamma_2'' \vdash \{\tau'\ x = v;\ e\} : \tau$ since $\Delta'' \hat{\subseteq} (X \cup \{\ell\}) \cdot \Gamma_1'' + \Gamma_2''$ we have the thesis.

(FIELD-ACCESS) We know that $x.f|\mu \rightarrow v|\mu$, hence $e = x.f$, $e' = v$. By Lemma 6.4.4 and rule (T-FIELD-ACCESS) we have $\Gamma'' \vdash x.f_i : \tau_i[\mathrm{M}]$ and $\Gamma'' \vdash x : C^{\mathrm{M}}$ such that $\Gamma'' \blacktriangleleft \Gamma_1$ and $\tau' = \tau_i[\mathrm{M}] \leq \tau$ with $\mathsf{fields}(C) = \tau_1\ f_1;\ \ldots \tau_n\ f_n;\ i \in 1..n$. We have two interesting cases:

- $\tau_i = D^{\mathsf{imm}}$
  By rule (T-VAR) we can derive $y :_{X'} D^{\mathsf{imm}} \vdash y : D^{\mathsf{imm}}$. Since $\tau = \tau_i[\mathrm{M}] = D^{\mathsf{imm}}$ we know that to $\Gamma'' \vdash x.f_i : D^{\mathsf{imm}}$ is not applied rule (T-CAPS), so, if we apply to $y :_X D^{\mathsf{imm}} \vdash y : D^{\mathsf{imm}}$ the same non syntax-directed rules applied to $\Gamma'' \vdash x.f_i : D^{\mathsf{imm}}$ we obtain $y :_X D^{\mathsf{imm}} \vdash y : \tau$, where $X = \mathsf{coeff}(\Gamma_1, x)$. By rule (T-MEM) and (T-OBJ) we have $\Delta_1 + \cdots + \Delta_n \Vdash [v_1, \ldots, v_n]^C : C^{\mathrm{M}'}$ and $\Delta_i \Vdash v_i : \tau_i[\mathrm{M}']$ where $\mu(x) = [v_1, \ldots, v_n]^C$, $\mathsf{fields}(C) = \tau_1\ f_1;\ \ldots \tau_n\ f_n;$ and exists i such that $v_i = y$. Since $\tau_i = D^{\mathsf{imm}}$ we know $\tau_i[\mathrm{M}] = D^{\mathsf{imm}}$ and $\Gamma_i \Vdash y : D^{\mathsf{imm}}$.By rule (T-IMM-REF) we obtain $\Delta_i = y :_{\{\ell\}} D^{\mathsf{imm}}$ with ℓ fresh. Since memory does not change applying rule (T-CONF) we obtain the thesis.
- $\tau_i = D^{\mathsf{mut}}$
  We know by Lemma 6.4.4 and rule (T-VAR) that $x :_{X''} \tau'' \vdash x : \tau''$ and $\Gamma'' = x :_{X'} C^{\mathrm{M}'}$. We can apply rule (T-VAR) and the same non syntax-directed rules applied to x on y to obtain $y :_{X'} \tau_i[\mathrm{M}'] \vdash y : \tau_i[\mathrm{M}]$. If we apply the same non syntax-directed rules applied to $\Gamma'' \vdash x.f_i : \tau_i[\mathrm{M}]$ we have $y :_X \tau_i[\mathrm{M}''] \vdash y : \tau$. We know $\mathsf{modif}(\Gamma, x) = \mathsf{modif}(y :_X \tau_i[\mathrm{M}''], y) = \mathrm{M}''$ and $\mathsf{coeff}(\Gamma_1, x) = X$. By rule (T-MEM) and (T-OBJ) we have $\Delta_1 + \cdots + \Delta_n \Vdash [v_1, \ldots, v_n]^C : C^{\mathrm{M}''}$ and $\Delta_i \Vdash v_i : \tau_i[\mathrm{M}'']$ where $\mu(x) = [v_1, \ldots, v_n]^C$, $\mathsf{fields}(C) = \tau_1\ f_1;\ \ldots \tau_n\ f_n;$ and exists i such that $v_i = y$. By rule (T-IMM-REF) we obtain $\Delta_i = y :_{\{\ell\}} \tau_i[\mathrm{M}'']$ with ℓ fresh. Since memory does not change applying rule (T-CONF) we obtain the thesis.

(CTX) We have $e = \mathcal{E}[e_1]$ and $e' = \mathcal{E}[e_1']$ and $e_1|\mu \rightarrow e_1'|\mu'$ and $\Gamma_1 \vdash \mathcal{E}[e_1] : \tau$.
We prove the thesis by induction on the evaluation context $\mathcal{E}$. We show only the relevant cases:

- $\mathcal{E} = [\ ]$ We know that $\Gamma \vdash \mathcal{E}[e_1]|\mu : \tau$ and that $\mathcal{E}[e_1] = e_1$, so $\Gamma \vdash e_1|\mu : \tau$. Applying the primary induction hypothesis to $e_1|\mu \rightarrow e_1'|\mu'$ and $\Gamma \vdash e_1|\mu : \tau$ and by knowing that $\mathcal{E}[e_1'] = e_1'$ we obtain the thesis.

- $\mathcal{E} = \mathcal{E}'.f = e'$ By Lemma 6.4.4 we know that exists a context Γ_1' such that $\Gamma_1' \blacktriangleleft \Gamma_1$ and a derivation $\mathcal{D}' : \Gamma_1' \vdash \mathcal{E}[e_1] : \tau'$ ending with a syntax directed rule. We know that the last applied rule in $\mathcal{D}'$ must

be (T-ASSIGN), so we know $\Gamma_1' = \Gamma_1'' + \Gamma_2''$ such that $\Gamma_1'' \vdash \mathcal{E}'[e_1] : C^{\mathsf{mut}}$ and $\Gamma_2'' \vdash e' : \tau_i$ with $\mathsf{fields}(C) = \tau_1\ f_1\mathbf{;}\ \ldots\ \tau_n\ f_n\mathbf{;}$ and $i \in 1..n$. We can apply rule (T-CONF) to obtain $\Gamma_1'' + \Gamma_2 \vdash \mathcal{E}'[e_1]||\mu : C^{\mathsf{mut}}$. By (CTX) we have $\mathcal{E}'[e_1]||\mu \to \mathcal{E}'[e_1']||\mu'$. By secondary induction hypothesis on this and on $\Gamma_1'' + \Gamma_2 \vdash \mathcal{E}'[e_1]||\mu : C^{\mathsf{mut}}$ we have that $\Theta' \vdash \mathcal{E}'[e_1']||\mu' : C^{\mathsf{mut}}$ such that

- $(\mathsf{erase}(\Gamma_1'' + \Gamma_2) + \mathsf{erase}(\Theta')) \restriction \mathsf{erase}(\Gamma_1'' + \Gamma_2) = \mathsf{erase}(\Gamma_1'' + \Gamma_2)$,
- for all $x \in \mathsf{dom}(\Gamma_1'' + \Gamma_2)$, $\mathsf{modif}(\Gamma_1'' + \Gamma_2, x) \leq \mathsf{modif}(\Theta', x)$

By rule (T-CONF) we have $\Theta' = \Delta_1' + \Gamma_2'$ such that $\Delta_1' \vdash \mathcal{E}'[e_1'] : \tau'$ and $\Gamma_2' \vdash \mu'$. By rule (T-ASSIGN) we have $\Delta_1' + \Delta_2 \vdash \mathcal{E}[e_1'] : \tau'$. We can prove that

- for all $x \in \mathsf{dom}(\Gamma_1') \cap \mathsf{dom}(\Delta_1' + \Delta_2)$, $\mathsf{modif}(\Gamma_1', x) \leq \mathsf{modif}(\Delta_1' + \Delta_2, x)$.

Applying the same non-syntax-directed rules applied to $\Gamma_1' \vdash \mathcal{E}[e_1] : \tau'$ and rule (T-CONF), by Lemma 6.4.5 we obtain the thesis.

- $\mathcal{E} = \{\tau_x\ x = \mathcal{E}'\mathbf{;}\ e'\}$ By Lemma 6.4.4 we know that exists a context Γ_1' such that $\Gamma_1' \blacktriangleleft \Gamma_1$ and a derivation $\mathcal{D}' : \Gamma_1' \vdash \mathcal{E}[e_1] : \tau'$ ending with a syntax directed rule. We know that the last applied rule in $\mathcal{D}'$ must be (T-BLOCK), so we know $\Gamma_1' = ((X \cup \{\ell\}) \cdot \Gamma_1'') \oplus \Gamma_2''$ such that $\Gamma_1'' \vdash \mathcal{E}'[e_1] : \tau_x$ and $\Gamma_2'', x :_X \tau_x \vdash e' : \tau_i$. We can apply rule (T-CONF) to obtain $\Gamma_1'' + \Gamma_2 \vdash \mathcal{E}'[e_1]||\mu : \tau_x$. By (CTX) we have $\mathcal{E}'[e_1]||\mu \to \mathcal{E}'[e_1']||\mu'$. By secondary induction hypothesis on this and on $\Gamma_1'' + \Gamma_2 \vdash \mathcal{E}'[e_1]||\mu : \tau_z$ we have $\Theta' \vdash \mathcal{E}'[e_1']||\mu' : \tau_z$ such that
  - $(\mathsf{erase}(\Gamma_1'' + \Gamma_2) + \mathsf{erase}(\Theta')) \restriction \mathsf{erase}(\Gamma_1'' + \Gamma_2) = \mathsf{erase}(\Gamma_1'' + \Gamma_2)$,
  - for all $x \in \mathsf{dom}(\Gamma_1'' + \Gamma_2)$, $\mathsf{modif}(\Gamma_1'' + \Gamma_2, x) \leq \mathsf{modif}(\Theta', x)$

  By rule (T-CONF) we have $\Theta' = \Delta_1' + \Gamma_2'$ such that $\Delta_1' \vdash \mathcal{E}'[e_1'] : \tau'$ and $\Gamma_2' \vdash \mu'$. By rule (T-BLOCK) we have $((X \cup \{\ell\}) \cdot \Delta_1') \oplus \Gamma_2'' \vdash \mathcal{E}[e_1'] : \tau'$. We can prove that
  - for all $x \in (\mathsf{dom}(\Gamma_1') \cap \mathsf{dom}(((X \cup \{\ell\}) \cdot \Delta_1') \oplus \Gamma_2''))$, $\mathsf{modif}(\Gamma_1', x) \leq \mathsf{modif}(((X \cup \{\ell\}) \cdot \Delta_1') \oplus \Gamma_2'', x)$

  Applying the same non-syntax-directed rules applied to $\Gamma_1' \vdash \mathcal{E}[e_1] : \tau'$ and rule (T-CONF), by Lemma 6.4.5 we obtain the thesis.

□

We now focus on properties of the memory ensured by this extended type system. First of all, we prove two lemmas characterising how the typing of memory propagates type modifiers. Recall that $\triangleright_\mu$ denotes the reachability relation in memory μ (Definition 6.2.6).

LEMMA 6.4.8 : If $\Gamma \vdash \mu$ and $x \triangleright_\mu y$, then

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

- $\mathsf{modif}(\Gamma, x) = \mathsf{imm}$ implies $\mathsf{modif}(\Gamma, y) = \mathsf{imm}$.

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

CASE $y = x$ The thesis trivially holds.

CASE $\mu(x) = [v_1, \ldots, v_n]^C$, $z = v_i$ FOR SOME $i \in 1..n$ AND $z \triangleright_\mu y$. Since $\Gamma \vdash \mu$, inverting rule (T-MEM), we have $\Delta \Vdash [v_1, \ldots, v_n]^C : C^{\mathrm{M}}$ for some Δ with $\Gamma = \Gamma' + \{\ell\} \cdot \Delta$ and $\mathrm{M} = \mathsf{modif}(\Gamma, x)$. Inverting rule (T-OBJ) and either (T-REF) or (T-IMM-REF), we have that $z :_X \tau_i[\mathrm{M}] \Vdash z : \tau_i[\mathrm{M}]$, with $\Delta = \Delta', z :_{\tau_i[\mathrm{M}]} X$ and $\mathsf{fields}(C) = \tau_1\, f_1\, ; \ldots \tau_n\, f_n\, ;$. Since $z \in \mathsf{dom}(\mu)$, τ_i is of shape $C'^{\mathrm{M}'}$. We split cases on M'.

- If $\mathrm{M}' = \mathsf{imm}$, then $\tau_i[\mathrm{M}] = \mathsf{imm}$, hence $\mathsf{modif}(\Gamma, z) = \mathsf{imm}$ and so, by induction hypothesis, we get $\mathsf{modif}(\Gamma, y) = \mathsf{imm}$ as needed.
- If $\mathrm{M}' = \mathsf{mut}$, then $\tau_i[\mathrm{M}] = \mathrm{M}$, hence $\mathsf{modif}(\Gamma, z) = \mathrm{M}$ and so the thesis follows by induction hypothesis.

□

LEMMA 6.4.9 : If $\Gamma \vdash \mu$, then, for all $x, y \in \mathsf{dom}(\mu)$, $\mathsf{coeff}(\Gamma, x) = \mathsf{coeff}(\Gamma, y)$ implies $\mathsf{modif}(\Gamma, x) = \mathsf{modif}(\Gamma, y)$.

In this refined setting, the definition of the sharing relation needs to take into account modifiers. Indeed, if intuitively two references are in sharing when a mutation of either of the two affects the other, then no sharing should be propagated through immutable references. To do so, we need to assume a well-typed memory in order to know modifiers of references.[7]

DEFINITION 6.4.10 (Sharing in memory with modifiers): The *sharing relation* in memory $\Gamma \vdash \mu$, denoted by $\bowtie_{\Gamma,\mu}$, is the smallest equivalence relation on $\mathsf{dom}(\mu)$ such that:

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

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

PROPOSITION 6.4.11 : If $\Gamma \vdash \mu$, then $x \bowtie_{\Gamma,\mu} y$ iff $\mathsf{coeff}(\Gamma, x) = \mathsf{coeff}(\Gamma, y)$, for all $x, y \in \mathsf{dom}(\mu)$.

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

THEOREM 6.4.12 (Capsule expression): If $\Gamma \vdash e|\mu : C^{\mathsf{caps}}$, and $e \mid \mu \rightarrow^\star y \mid \mu'$, then there exists Γ' such that $\Gamma' \vdash \mu'$, $\Gamma \leq \Gamma'$ and, for all $x \in \mathsf{dom}(\mu)$, $x \bowtie_{\Gamma',\mu'} y$ implies $\mathsf{modif}(\Gamma, x) \leq \mathsf{modif}(\Gamma', x) \neq \mathsf{mut}$.

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

Proof: By Theorem 6.4.2, we get $\Delta \vdash y|\mu' : C^{\text{caps}}$ with $\Gamma \leq \Delta$. By inverting rule (T-CONF), we get $\Delta_1 \vdash y : C^{\text{caps}}$ and $\Delta_2 \vdash \mu'$, with $\Delta = \Delta_1 + \Delta_2$ and $\Gamma \leq \Delta_2$, as $\mathsf{modif}(\Delta_2, z) = \mathsf{modif}(\Delta, z)$ for all $\in \mathsf{dom}(\Delta)$. Since $y \in \mathsf{dom}(\mu')$, it cannot have modifier caps, hence $\Delta_1 \vdash y : C^{\text{caps}}$ holds by rule (T-PROM) or (T-SUB). This implies $\Delta_1 = \emptyset \cdot \Delta', y :_{\{\mathrm{R}\}} C^{\sigma}$ and so $\mathsf{modif}(\Delta, y) = \mathsf{modif}(\Delta_2, y) = \mathsf{modif}(\Delta_1, y) = \sigma$. Set $\Gamma' = \Delta_2$. By Proposition 6.4.11 and Lemma 6.4.9, $x \bowtie_{\Gamma',\mu'} y$ implies $\mathsf{modif}(\Gamma', x) = \sigma$, thus we get $\mathsf{modif}(\Gamma, x) \leq \mathsf{modif}(\Delta, x) = \mathsf{modif}(\Gamma', x) \neq \mathsf{mut}$, hence the thesis. □

It is important to notice that the notion of capsule expression in Theorem 6.4.12 is different from the previous one (Definition 6.2.4), as we now have imm references. In particular, the previous notion prevented any access to the reachable object graph of the result from free variables, since, without modifiers, any access to a portion of memory can modify it. Here, instead, this is no longer true, hence the notion of capsule allows mutable references to access the reachable object graph of the result of a capsule expression, but only through imm references. Indeed, if two references access the same non-imm reference, they are necessarily in sharing, as shown below.

PROPOSITION 6.4.13 : Let $\Gamma \vdash \mu$. If $x \triangleright_{\mu} z$ and $y \triangleright_{\mu} z$ and $\mathsf{modif}(\Gamma, z) \neq \mathsf{imm}$, then $x \bowtie_{\Gamma,\mu} y$.

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

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

CASE $\mu(x) = [v_1, \ldots, v_n]^C$, $x' = v_i$ FOR SOME $i \in 1..n$ AND $x' \triangleright_{\mu} z$
We know that $\mathsf{modif}(\Gamma, x)$,
$\mathsf{modif}(\Gamma, x') \leq \mathsf{mut}$ because $\mathsf{modif}(\Gamma, x') = \mathsf{imm}$ (or $\mathsf{modif}(\Gamma, x) = \mathsf{imm}$) would imply $\mathsf{modif}(\Gamma, z) = \mathsf{imm}$, by Lemma 6.4.8, which is a contradiction. Therefore, by Definition 6.4.10, we have $x \bowtie_{\Gamma,\mu} x'$ and by induction hypothesis, we get $x' \bowtie_{\Gamma,\mu} z$; then we get $x \bowtie_{\Gamma,\mu} z$ by transitivity of $\bowtie_{\Gamma,\mu}$.

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

COROLLARY 6.4.14 : If $\Gamma \vdash e|\mu : C^{\text{caps}}$, and $e \mid \mu \rightarrow^{\star} y \mid \mu'$, then there exists Γ' such that $\Gamma' \vdash \mu'$, $\Gamma \leq \Gamma'$ and, for all $x \in \mathsf{dom}(\mu)$, $\mathsf{modif}(\Gamma', x) = \mathsf{mut}$ and $x \triangleright_{\mu'} z$ and $y \triangleright_{\mu'} z$ imply $\mathsf{modif}(\Gamma', z) = \mathsf{imm}$.

Proof: By Theorem 6.4.12, we get $\Gamma' \vdash \mu'$ and $x \bowtie_{\Gamma',\mu'} y$ imply $\mathsf{modif}(\Gamma', x) \neq \mathsf{mut}$. Suppose $\mathsf{modif}(\Gamma', z) \neq \mathsf{imm}$, then, by Proposition 6.4.13, we get $x \bowtie_{\Gamma',\mu'} y$, hence $\mathsf{modif}(\Gamma', x) \neq \mathsf{mut}$, which contradicts the hypothesis. Therefore, $\mathsf{modif}(\Gamma', z) = \mathsf{imm}$. □

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

LEMMA 6.4.15 : If $\Gamma \vdash e|\mu : \tau$, and $\mathsf{modif}(\Gamma, x) = \mathsf{imm}$, and $e|\mu \longrightarrow e'|\mu'$, then $\mu(x) = \mu'(x)$.

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

Thanks to Lemma 6.4.8, we can show that the reachable object graph of an imm reference contains only imm references. Hence, by the above lemma we can characterise imm references as follows:

THEOREM 6.4.16 (Immutable reference): If $\Gamma \vdash e|\mu : \tau$, $\mathsf{modif}(\Gamma, x) = \mathsf{imm}$, and $e \mid \mu \rightarrow^\star e' \mid \mu'$, then $x \triangleright_\mu y$ implies $\mu(y) = \mu'(y)$.

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

CASE $e \mid \mu = e' \mid \mu'$ The thesis trivially holds.

CASE $e \mid \mu | e_1 \mid \mu_1 \longrightarrow \rightarrow^\star | e' \mid \mu'$ Since $\mathsf{modif}(\Gamma, x) = \mathsf{imm}$, for all y such that $x \triangleright_\mu y$, by Lemma 6.4.8 $\mathsf{modif}(\Gamma, y) = \mathsf{imm}$, hence, by Lemma 6.4.15, $\mu_1(y) = \mu(y)$. Therefore, it is easy to check that $x \triangleright_\mu y$ implies $x \triangleright_{\mu_1} y$. By Theorem 6.4.2, $\Delta \vdash e_1|\mu_1 : \tau$ and $\mathsf{modif}(\Gamma, x) \leq \mathsf{modif}(\Delta, x)$, hence $\mathsf{modif}(\Delta, x) = \mathsf{imm}$. Then, by induction hypothesis, $\mu'(y) = \mu_1(y)$, hence the thesis. □

6.5 Expressive power

We discuss the expressive power of the type-and-coeffect system in Section 6.4, comparing it with the two most closely related proposals by Gordon et al. [35] and Pony [19, 20]. The takeaway is that our promotion mechanism is much more powerful than their recovery, since sharing is taken into account; on the other hand, the expressive power allowed by some of their annotations on fields is beyond the scope of this thesis.

We assume a syntax enriched by the usual programming constructs.

Before the work by Gordon et al. [35], the capsule property was only ensured in simple situations, such as using a primitive deep clone operator, or composing subexpressions with the same property. The type system by Gordon et al. [35] has been an important step, being the first to introduce *recovery*. That is, this type system contains two typing rules which allow recovering isolated[8] or immutable references from arbitrary code checked in contexts containing only isolated or immutable variables. Such rules are rephrased below in our style for better comparison.

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

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

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

As the reader can note, this is exactly in the spirit of coeffects, since typechecking also takes into account the way the *surrounding context* is used. By these rules Gordon et al. [35] typechecks, e.g., the following examples, assuming the language has threads with a parallel operator:

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

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

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

An isolated object can be mutated[9], and then isolation can be recovered, since the context only contains isolated or immutable references.

In Pony [19, 20], the ideas of Gordon et al. [35] are extended to a richer set of modifiers. In their terminology `val` is immutable, `ref` is mutable, `box` is read-only. An ephemeral isolated reference `iso^` is similar to a caps reference in our calculus, whereas non ephemeral `iso` references are more similar to the isolated fields discussed below. Finally, `tag` only allows object identity checks and asynchronous method invocation, and `trn` (transition) is a subtype of `box` that can be converted to `val`, providing a way to create values without using isolated references. The last two modifiers have no equivalent in what proposed by Gordon et al. [35] or our work.

The type-and-coeffect-system in Section 6.4 shares with the work by Gordon et al. [35] and Pony the modifiers mut, imm, read, and caps with their subtyping relation, a similar operation to combine modifiers, and the key role of recovery. However, rule (T-PROM) is much more powerful than the recovery rules reported above, which definitely forbid read and mut variables in the

8 Their terminology for capsule.

9 We say that the capsule is *opened*, see in the following.

context. Rule (T-PROM), instead, allows such variables when they are not connected to the final result, as smoothly derived from coeffects which compute sharing. For instance, with the classes of Example 6.2.2, the following two examples would be ill-typed in and Pony:

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

Below is the corresponding Pony code.

```
class B
  var f: U64
  new create(ff:U64) => f=ff
class C
  var f1: B
  var f2: B
  new create(ff1: B ref, ff2: B ref) =>  f1=ff1; f2=ff2
var x: B ref = ...
var y: B ref = ...
var es1: C iso =
  recover iso var z = B(2); x.f1=y; C(z,z) end//ERROR
var es2: C iso =
  recover iso var z = B(y.f=y.f+1); C(z,z) end//ERROR
```

A comparison on a more involved example, notably our running example, is provided later on.

Another relevant difference with Gordon et al. [35] and Pony is that they support isolated fields, using an ad-hoc semantics, called *destructive read*, see also what done by Boyland [14]. Gordon et al. [35] allow to an isolated field to be read only by a command `x = consume(y.f)`, assigning the value to `x` and updating the field to `null`.

Pony supports the command `(consume x)`, with the semantics that the reference becomes empty. Since fields cannot be empty, they cannot be arguments of `consume`. By relying on the fact that assignment returns the left-hand side value, in Pony one writes `x=y.f=(consume z)`, with `z` isolated. In this way, the field value is assigned to `x`, and the field is updated to a new isolated reference.

We prefer to avoid destructive reads since they can cause subtle bugs [33]. As a consequence, in our approach the caps modifier cannot be applied to fields. Isolated fields with destructive reads could be encoded by adding a primitive `IsoBox<T>` type with `set` and `get` methods, the former allowing to store a caps reference in the `IsoBox`, and the latter retrieving the stored reference and marking the content as `removed`.

Let us add to class `A` of Example 6.4.1 the method `nonMix` that follows:

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

Consider the following code:

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

```
  // a1.mix(a2).clone().mix(a2) // (2)
  // a1.nonMix(a2) // (3)
}
```

The corresponding Pony code is as follows:

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

As in our approach, Pony discriminates line (1) from line (2), causing code to be well-typed and ill-typed, respectively. However, to do so, Pony needs a modifier `iso^` in the return type of `clone`, whereas, as noted after the code of Example 6.4.1, in our approach the return type of `clone` can be mut, since the fact that there is no connection between the result and `this` is expressed by the coeffect. Moreover, to be able to obtain an `iso` from the `clone` method, Pony needs to insert explicit `recover` instructions. In the case of class `A` where the field is an object, Pony needs to explicitly use `consume` to ensure uniqueness, whereas in our approach promotion takes place implicitly and uniqueness is ensured by linearity. Finally, Pony rejects line (3) as well, whereas, in our approach, this expression is correctly recognized to be a capsule, since the external variable `a1` is modified, but not connected to the final result.

We end the section with an example illustrating how our type system can prevent sharing of parameters, something not possibile in the work by Gordon et al. [35] and Pony.

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

The method takes two teams, `t1` and `t2`, as parameters. Both want to add a reserve player from their respective lists `p1` and `p2`, sorted with best players first. However, to keep the game fair, the two reserve players can only be added if they have the same skill level. The sharing coeffects express that each team can only add players from its list of reserve players.

7

Related work

Our major sources of inspiration have been Petricek's thesis [50, 51, 52], introducing the notion of coeffect, the Granule project, see https://granule-project.github.io, showing how to design a fully-fledged language equipped with a graded type system as shown by Orchard, Liepelt, and Harley Eades III [49], and Choudhury et al. [18], introducing instrumented semantics in order to prove resource-aware soundness.

We describe the first two and the third, together with other relevant contributions, in Section 7.1 and Section 7.2, respectively. In Section 7.3, instead, we provide an overview of the wide literature about type systems for tracking sharing and/or immutability.

7.1 Resource-aware type systems

Coeffects were firstly introduced by Petricek, Orchard, and Mycroft [52] and further analyzed by Petricek, Orchard, and Mycroft [51]. In particular, Petricek, Orchard, and Mycroft [51] develop 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 on two opposite instances: structural (per-variable) and flat (whole context) coeffects, identified by specific choices of context shapes. The authors present many examples (liveness, dataflow, dynamic scoping).

Most of the subsequent literature on coeffects focuses on structural ones, as those from Chapter 3 to Chapter 5, for which there is a clear algebraic description in terms of semirings. This was first noticed by Brunel et al. [16], who developed a framework of structural coeffects for a functional language. His approach is inspired by a generalization of the exponential modality of linear logic, see, e.g., Breuvart and Pagani [15]. That is, as discussed in Chapter 1, the distinction between linear and unrestricted variables of linear systems is generalized to have variables decorated by coeffects, or grades, that determine how much they can be used.

Other graded type systems are explored by Abel and Bernardy [2], Atkey [5], and Ghica and Smith [31], also combining coeffects with other programming features, such as computational effects, as shown by Dal Lago and Gavazzo [26], Gaboardi et al. [30], and Torczon et al. [54] and Orchard, Liepelt, and Harley Eades III [49], dependent types, as shown by Atkey [5], Choudhury et al. [18], and McBride [46], and polymorphism, as shown by Abel and Bernardy [2].

In all these papers, the process of tracking usage through grades is a powerful method of instrumenting type systems with analyses of irrelevance and linearity that have practical benefits like erasure of irrelevant terms (resulting in speed-up) and compiler optimizations (such as in-place update).

The type system presented in Chapter 5 may look similar to the one presented by McBride [46], except that grades are assigned to typing judgements instead of types, but there are relevant differences. The first one is typing rule for lambda abstraction. Below we report both the rules where in our rule we do not consider recursion for the sake of simplicity:

$$\text{(Lam)}\ \frac{\Gamma, x :_{r_1} \tau_1 \vdash e : \tau_2^{r_2}}{r \cdot \Gamma \vdash \lambda x.e : (\tau_1^{r_1} \to \tau_2^{r_2})^r}$$

$$\text{(Lam-McBride)}\ \frac{\Gamma, x :_{r \cdot s} \tau_1 \vdash e :_r \tau_2}{\Gamma \vdash \lambda x.e :_r (x :_s \tau_1) \to \tau_2}$$

We can notice that in (Lam) the grade on the type of the lambda expression is free, whereas the grade on the conclusion judgment in (Lam-McBride) is not. Not only that, but the grade on the consequence judgement interact with the grade of the input in the premise context. The fact is that in our system grades can be nested, whereas in McBride's one grades are combined; the structure of types is different. In our system arrow types, for example, have a grade representing how many times the function can be used, one counting how many times the input is used in the body and one grade counting how many times the output can be used, whereas McBride's arrow types annotates only the usage of input in the body. Another important difference is that in the system proposed by McBride [46] substitution is not sound, as observed by Atkey [5], whereas we proved a substitution lemma for value in Lemma 5.2.8.

As said above, Granule is a fully-fledged functional programming language following the principles of resource-aware type systems, as described by and Orchard, Liepelt and Harley Eades III [49].Granule is equipped with graded types, where different kinds of grades can be used at the same time, including naturals for exact usage, security levels, intervals, infinity, and products of coeffects. We owe to Granule, first of all, the overall aim of providing foundations for the design of a resource-aware programming language. We focused on the object-oriented, rather than functional, paradigm. Moreover, we were inspired by Granule for the idea of allowing different kinds of grades to coexist; we pushed forward this approach to have a grade algebra which is not fixed, but extendable by the programmer with user-defined grades.

Hanukaev and Harley Eades III [38] present a type system inspired by Adjoint Logic, called Glad, offering a smooth way of combining an arbitrary number of logics with different substructural rules and different resource semirings, identified by *modes*. This idea has many similarities with the construction in Section 4.2. Notably, the coexisting modes are defined by a functor from a preordered set of indexes (corresponding to our kinds with their partial order) to the category of modes (corresponding to our category of grade algebras). The construction is different, notably, to combine elements of different modes κ and μ, they require $\kappa \leq \mu$, and apply the corresponding morphism to the

element of mode κ. In Section 4.2, instead, our aim is to always be able to combine grades of different kinds, by mapping both in grades of their (assumed to exist) least common ancestor. We leave to further work a detailed comparison.

McBride [46] and Wood and Atkey [56] firstly observed, as already mentioned, that contexts in a structural coeffect system form a module over the semiring of grades, event though they do not use this structure in its full generality, restricting themselves to free modules, that is, to structural coeffect systems.

7.2 Resource-aware semantics

The main source of inspiration for our resource-aware semantics and resource-aware soundness results has been what done by Choudhury et al. [18]. In this work, the authors develop GRAD, a graded dependent type system that includes functions, tensor products, sums, and a unit type. They define an instrumented operational semantics which tracks usage of resources, and prove that the graded type system is sound with respect to such instrumented semantics. Moreover, they derive from this theorem several useful properties, such as standard type soundness, non-interference of irrelevant resources in computation and single pointer property for linear resources.

The lambda-calculus with molteplicities, proposed in an earlier paper by Boudol [13], can be seen as providing a kind of resource-aware semantics. In this work, inspired by the encoding of lazy λ-calculus in π-calculus as done by Milner [48], variable occurrences are replaced once at a time when needed, as in our instrumented semantics. The environment associates to variables a multiset (bag) of values and, each time a variable occurrence has to be replaced, a value is non-deterministically chosen from the bag, so reduction is stuck if the bag is empty. Differently from our approach, the environment is part of the syntax of terms, as in explicit substitutions by Abadi et al. [1] and Curien, Hardin, and Lévy [22]. The number of values in the bag corresponds to a natural number grade in our framework.

Laird et al. [41] have a different approach to resource-aware semantics. Here the semantics is described by reduction steps weighted on elements of a semiring, so they are used to weight reduction paths. The resources, rather than free variables, are time or space of execution. In the case of a non-deterministic language, by taking as a semiring booleans with the usual operations, we can check whether a program can be reduced to a given natural number; the boolean annotation expresses whether the reduction step does or not what we expect. If we take as semiring natural numbers, instead, then we can count how many reduction paths exists for a program to a given natural number; the annotation keeps track of the paths that do what we expect. Also in this work, even though the notion of resource is different, we can compare programs not only with respect to their input-output behaviour, but also by considering how they consume resources.

7.3 Sharing

The literature on type systems controlling sharing and mutation is huge. We already provided a detailed comparison of our approach with the two mostly related proposals, the one proposed by Gordon et al. [35] and Pony [19, 20], in Section 6.5.

The approach based on modifiers is extended by Castegren and Wrigstad [17] and Haller and Odersky [37] to compositions of one or more *capabilities*. The modes of the capabilities in a type control how resources of that type can be aliased. The compositional aspect of capabilities is an important difference from modifiers, as accessing different parts of an object through different capabilities in the same type gives different properties. By using capabilities it is possible to obtain an expressivity similar to the type system in Chapter 6, although with different sharing notions and syntactic constructs. For instance, the *full encapsulation* notion by Haller and Odersky [37], apart from the fact that sharing of immutable objects is not allowed, is equivalent to our caps guarantee. Their model has a higher syntactic/logic overhead to explicitly track regions. As for all work preceding Gordon et al. [35], objects need to be born `unique` and the type system permits manipulation of data preserving their uniqueness. With recovery/promotion, instead, we can use normal code designed to work on conventional shared data, and then recover uniqueness.

An approach alternative to modifiers to restrict the usage of references is that of *ownership*, based on *enforcing invariants* rather than deriving properties. We refer to the recent work of Milano, Turcotti, and Myers [47] for an up-to-date survey. The Rust language, considering its "safe" features [40], belongs to this family as well, and uses ownership for memory management. In Rust, all references which support mutation are required to be affine, thus ensuring a unique entry point to a portion of mutable memory. This relies on a relationship between linearity and uniqueness recently clarified by Marshall, Vollmer, and Orchard [45], which proposes a linear calculus with modalities for non-linearity and uniqueness with a somewhat dual behaviour. In our approach, instead, the capsule concept models an efficient *ownership transfer*. In ownership, when an object x is "owned" by y, it remains always true that y can only be accessed through x, whereas the capsule notion is dynamic: a capsule can be "opened", that is, assigned to a standard reference and modified, since we can always recover the capsule guarantee. We also mention that, whereas in Chapter 6 all properties are *deep*, that is, inherited by the reachable object graph, most ownership approaches allows one to distinguish subparts of the reachable object graph that are referred to but not logically owned. This viewpoint has some advantages, for example Rust uses ownership to control deallocation without a garbage collector.

An approach based on regions is proposed by Arvidsson et al. [4]. In this paper, memory is divided in independent regions, which can be open, closed or suspended. The current state of the region characterizes its allowed usage. For example, objects of an open region are permitted to reference objects in suspended regions. Types are decorated with capabilities similar to the modifi-

ers presented in Chapter 6. A difference is that there is a specific capability for immutable references of a suspended region, that is, in a sense, a temporary capability. The type of an expression is given from the point of view of the currently active region and this is obtained by viewpoint adaptation. An interesting feature is that at a given time only one region is mutable, that is, we have a *single window of mutability*.

Reachability types, proposed by Bao et al. [6], qualify types with sets of reachable variables and guarantee separation if two terms have disjoint qualifiers, similarly to what we achieve in Chapter 6 with set of links and transitive closure. Wei et al. [55] describe a different version, where, instead of always tracking the transitive closure of reachable variables, only variables reachable in a single step are tracked, and transitive closures computed only when necessary, thus preserving chains of reachability over known variables that can be refined using substitution.

8

Conclusion

The overall outcome of this thesis are design guidelines and formal techniques to keep track of the usage of resources in programming languages.

A substantial part of the work has been devoted to the object-oriented paradigm, notably to Java-like languages. We have designed a resource-aware extension of a paradigmatic Java-like calculus, and proved resource-aware soundness, meaning that the type system guarantees the possibility of using resources only when/how they are available. Moreover, we have described how to give to the programmer the capability to define her/his own grades, by relying on a formal construction for combining grades of different kinds.

In Chapter 5 we have applied the approach, instead, to a functional language, posing the new challenges of higher-order functions and recursive types. Moreover, we have defined the resource-aware semantics in big-step style, thus avoiding the overhead of grade annotations. We were able to prove resource-aware soundness, hardly even expressed in the big-step case, by relying on recently introduced generalized coinductive techniques.

In Chapter 6 we have considered a difficult case of usage of resources, that is, taking into account sharing in an imperative language. We have shown that this can be actually tracked by coeffects, by moving from the structural ones to a more general framework.

We identify the following key novel contributions of the thesis.

SMOOTH RESOURCE-AWARENESS We provide guidelines to add resource-awareness that are very general and mostly independent from the language. As in many papers, such generality is achieved by being parametric on an arbitrary grade algebra, and by considering, as we do in Chapter 5, powerful enough language features. However, differently from previous work, we add resource-awareness without requiring ad-hoc changes to the underlying language. The advantage is to keep, as far as possible, the language unaffected from the point of view of the programmers.

GRADES IN OO PARADIGM Whereas graded type systems for functional calculi have been widely studied in literature, there was, as far as we know, no previous attempt at investigating resource-awareness in the context of the object-oriented paradigm, followed by many of the most popular languages used today. Some of our solutions seem more adequate in that context, e.g., to have once-graded types and no boxing/unboxing. An important observation

emerged from our work is that, as discussed in Section 5.4, record/object calculi, where fields can be discarded at runtime, whereas object construction happens by their sequential evaluation, seems to be hardly compatible with non-affine grades. In future work we plan to investigate this interesting problem.

EXTENSIBLE GRADES As already mentioned, we extended the Granule approach by Orchard, Liepelt and Harley Eades III [49], supporting *heterogeneous* grades, in the sense that grades of different kinds are simultaneously available to the programmer, by allowing the grade algebra to be, rather than fixed once and for all, *extensible* by the programmer, similarly to what happens, e.g., with Java exceptions. The design and implementation of a real Java-like language are beyond the objectives of this thesis; however, we outline in Section 4.3 two approaches, which can be adopted in general to implement user-defined grades in a real language. That is, either grades are defined in a different, more appropriate, language, in our case a simple extension of the Java-like calculus with special classses; or they are encoded in the language itself, in our case by relying on Java generic interfaces and classes. In the first approach, we could even use a language with dependent types, to be able to also prove the required properties. The second approach could be achieved in different paradigms as well, e.g., in Haskell, by using type classes.

A prototype implementation of a coeffect checker for MiniJava is available, by Duso [29]. MiniJava is an imperative extension of the language by Bianchini et al. [8], and besides assignment includes additional language constructs. Java annotations are used to decorate coeffect classes and coeffect annotations for variables.

BIG-STEP SEMANTICS AND GRADES In Chapter 5 we present the first, at our knowledge, resource-aware semantics in big-step style. Besides the classical advantages, such as the higher level of abstraction, in big-step style we avoid the overhead of grade annotations in subterms. Note also that soundness expressed for big-step semantics implicitly turns out to be soundness-may, since the statement is that, for a well-typed term, there a derivable reduction judgment to a result (including divergence), see Theorem 5.3.1 and Theorem 5.3.7. In small-step style, instead, the standard proof technique by progress and subject reduction ensures soundness *must*, that is, that all reduction sequences do not get stuck; to express soundness-may an ad-hoc formulation is needed, see Theorem 3.4.2 and Corollary 3.4.9.

NON-STRUCTURAL EXAMPLE Finally, in Chapter 6 we show a non-trivial example of coeffects which can be considered *quasi-structural*. Indeed, coeffects cannot be computed per-variable (that is, the sum and scalar multiplication operators on coeffect contexts are *not* defined pointwise), but can still be expressed by annotations on single variables, differently from *flat* contexts [50, 51, 52] . This significant example justifies a framework in which coeffect contexts are modules, including structural coeffect systems as the simplest case. As future work, it could be interesting to further investigate such framework,

both at the level of meta-theory and of finding other significant examples.

Besides the specific points mentioned above, we identity the following particularly interesting directions for future work:

EXACT USAGE TRACKING 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, as shown in Section 5.4. However, in the current instrumented semantics, there is the requirement that needed resources should be *available* (otherwise reduction gets stuck), but the opposite constraint that resources should be *non-wasted* is not expressed. We plan to investigate what happens by introducing this constraint as well. In this way, we should be able to express a more powerful resource-aware soundness theorem: for well-typed programs, there is a reduction which, besided being non-stuck, is also *non-wasting*. So, in the case of affine grades, the initially available resources are all exhausted at the end. A possible way to obtain such a semantics is to consume exactly the required amount when a variable occurrence is encountered, and, when reducing a compound term, to reduce subterms in environments obtained by decomposing the original one, mimicking the type system.

FROM UNIQUENESS TO ASSUMPTION GRADES The recent work by Marshall, Vollmer, and Orchard [45] proposes a unified calculus and type system that incorporates linear, unique, and unrestricted types all at once, building on the linear lambda-calculus. In this way, the authors nicely clarify the relation between linearity and uniqueness, sometimes confused in previous literature: linearity means that some resource is used exactly once, whereas uniqueness means that there should be at most one reference to the resource. As graded types are a generalization of linearity, expressing that the resource should be used with a certain grade rather than exactly once, we believe it would be interesting to develop a similar generalization for uniqueness. Thinking to coeffects, where the idea is simpler to explain, when typechecking an expression e, each variable in the type-and-coeffect context should be annotated by *two* coeffects: the *usage* grade (the standard one), expressing how the variable is used in e, and the *assumption grade*, expressing the assumption e can make about how the variable can be used in a program context enclosing e. With the terminology of Marshall, Vollmer, and Orchard [45], the two grades track the future and past usage, respectively. For instance, if both standard and assumption grades are natural numbers with exact counting, if a variable has coeffects $1, 0$, then this means that x is used linearly in e, and, moreover, e can safely assume that x will not be used elsewhere, thus expressing a uniqueness property. We hope this abstract approach could be instantiated to characterize sharing/immutability properties in the imperative paradigm, as we did in a more ad-hoc way in Chapter 6.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In: *Journal of Functional Programming* 1.4 (1991), 375–416. DOI: 10.1017/S0956796800000186. Cited on p. 131.

[2] Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. In: *Proceedings of ACM on Programming Languages* 4.ICFP (2020), 90:1–90:28. DOI: 10.1145/3408972. Cited on pp. 7, 9–11, 23, 129.

[3] Davide Ancona, Francesco Dagnino, and Elena Zucca. Reasoning on Divergent Computations with Coaxioms. In: *Proceedings of ACM on Programming Languages* 1.OOPSLA (2017), 81:1–81:26. DOI: 10.1145/3133905. Cited on pp. 59, 76, 78, 79.

[4] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. Reference Capabilities for Flexible Memory Management. In: *Proceedings of ACM on Programming Languages* 7.OOPSLA2 (2023), pp. 1363–1393. DOI: 10.1145/3622846. Cited on p. 132.

[5] Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In: *IEEE Symposium on Logic in Computer Science, LICS 2018*. Edited by Anuj Dawar and Erich Grädel. ACM Press, 2018, pp. 56–65. DOI: 10.1145/3209108.3209189. Cited on pp. 7, 10, 129, 130.

[6] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: tracking aliasing and separation in higher-order functional programs. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–32. DOI: 10.1145/3485516. URL: https://doi.org/10.1145/3485516. Cited on p. 133.

[7] Riccardo Bianchini. Monitoring for Resource-Awareness. In: *Verification and Monitoring at Runtime Execution, VORTEX 2023*. Edited by Davide Ancona and Giorgio Audrito. ACM Press, 2023, pp. 13–16. DOI: 10.1145/3605159.3605856. Cited on p. 3.

[8] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. A Java-like calculus with heterogeneous coeffects. In: *Theoretical Computer Science* 971 (2023), p. 114063. DOI: https://doi.org/10.1016/j.tcs.2023.114063. Cited on pp. 3, 40, 136.

[9] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. A Java-like calculus with user-defined coeffects. In: *ICTCS'22 - Italian Conference on Theoretical Computer Science*. Edited by Ugo Dal Lago and Daniele Gorla. Vol. 3284. CEUR Workshop Proceedings. CEUR-WS.org, 2022, pp. 66–78. Cited on pp. 3, 40.

[10] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. Multi-graded Featherweight Java. In: *European Conference on Object-Oriented Programming, ECOOP 2023*. Edited by Karim Ali and Guido Salvaneschi. Vol. 263. LIPIcs. 756, 2023, 3:1–3:27. DOI: 10.4230/LIPIcs.ECOOP.2023.3. Cited on pp. 3, 40.

[11] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. Resource-Aware Soundness for Big-Step Semantics. In: *Proceedings of ACM on Programming Languages* 7.OOPSLA2 (2023), pp. 1281–1309. DOI: 10.1145/3622843. URL: https://doi.org/10.1145/3622843. Cited on p. 4.

[12] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, and Marco Servetto. Coeffects for sharing and mutation. In: *Proceedings of ACM on Programming Languages* 6.OOPSLA (2022), pp. 870–898. DOI: 10.1145/3563319. Cited on pp. 4, 96.

[13] Gérard Boudol. The Lambda-Calculus with Multiplicities (Abstract). In: *Concurrency Theory, CONCUR 1993*. Edited by Eike Best. Vol. 715. Lecture Notes in Computer Science. Springer, 1993, pp. 1–6. DOI: 10.1007/3-540-57208-2_1. Cited on pp. 2, 131.

[14] John Boyland. Semantics of Fractional Permissions with Nesting. In: *ACM Transactions on Programming Languages and Systems* 32.6 (2010). Cited on p. 126.

[15] Flavien Breuvart and Michele Pagani. Modelling Coeffects in the Relational Semantics of Linear Logic. In: *Computer Science Logic, CSL 2015*. Edited by Stephan Kreutzer. Vol. 41. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 567–581. DOI: 10.4230/LIPIcs.CSL.2015.567. Cited on pp. 116, 129.

[16] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A Core Quantitative Coeffect Calculus. In: *European Symposium on Programming, ESOP 2013*. Edited by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 351–370. DOI: 10.1007/978-3-642-54833-8_19. Cited on pp. 7, 10, 129.

[17] Elias Castegren and Tobias Wrigstad. Reference Capabilities for Concurrency Control. In: *European Conference on Object-Oriented Programming, ECOOP 2016*. Edited by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 5:1–5:26. Cited on p. 132.

[18] Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. A graded dependent type system with a usage-aware semantics. In: *Proceedings of ACM on Programming Languages* 5.POPL (2021), pp. 1–32. DOI: 10.1145/3434331. Cited on pp. 2, 7, 10, 16, 129, 131.

[19] Sylvan Clebsch. *'Pony': co-designing a type system and a runtime*. PhD thesis. Imperial College London, UK, 2017. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.769552. Cited on pp. 124, 125, 132.

[20] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In: *International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015*. Edited by Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela. ACM Press, 2015, pp. 1–12. Cited on pp. 124, 125, 132.

[21] Patrick Cousot and Radhia Cousot. Inductive Definitions, Semantics and Abstract Interpretations. In: *ACM Symposium on Principles of Programming Languages, POPL 1992*. Edited by Ravi Sethi. ACM Press, 1992, pp. 83–94. DOI: 10.1145/143165.143184. Cited on pp. 59, 76, 78.

[22] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence Properties of Weak and Strong Calculi of Explicit Substitutions. In: *J. ACM* 43.2 (1996), pp. 362–397. DOI: 10.1145/226643.226675. URL: https://doi.org/10.1145/226643.226675. Cited on p. 131.

[23] Francesco Dagnino. A Meta-theory for Big-step Semantics. In: *ACM Transactions on Computational Logic* 23.3 (2022), 20:1–20:50. DOI: 10.1145/3522729. Cited on pp. 32, 74, 78.

[24] Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. In: *Logical Methods in Computer Science* 15.1 (2019). DOI: 10.23638/LMCS-15(1:26)2019. Cited on pp. 59, 76, 78, 79.

[25] Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness Conditions for Big-Step Semantics. In: *European Symposium on Programming, ESOP 2020*. Edited by Peter Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 169–196. DOI: 10.1007/978-3-030-44914-8_7. Cited on pp. 32, 74.

[26] Ugo Dal Lago and Francesco Gavazzo. A relational theory of effects and coeffects. In: *Proceedings of ACM on Programming Languages* 6.POPL (2022), pp. 1–28. DOI: 10.1145/3498692. Cited on p. 129.

[27] Rocco De Nicola and Matthew Hennessy. Testing Equivalences for Processes. In: *Theoretical Computer Science* 34.1 (1984), pp. 83 –133. DOI: https://doi.org/10.1016/0304-3975(84)90113-0. Cited on pp. 32, 74.

[28] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In: *European Conference on Object-Oriented Programming, ECOOP 2007*. Edited by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer, 2007, pp. 28–53. Cited on pp. 23, 28.

[29] Giulio Duso. *Implementazione di controllo di tipi e coeffetti per MiniJava*. Available at https://github.com/DusoGiulio/Coeff_Inference/tree/main. MA thesis. Università del Piemonte orientale, 2023. Cited on p. 136.

[30] Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. Combining effects and coeffects via grading. In: *ACM International Conference on Functional Programming, ICFP 2016*. Edited by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM

Press, 2016, pp. 476–489. DOI: 10.1145/2951913.2951939. Cited on pp. 7, 10, 129.

[31] Dan R. Ghica and Alex I. Smith. Bounded Linear Types in a Resource Semiring. In: *European Symposium on Programming, ESOP 2013*. Edited by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 331–350. DOI: 10.1007/978-3-642-54833-8_18. Cited on pp. 7, 10, 129.

[32] Paola Giannini, Tim Richter, Marco Servetto, and Elena Zucca. Tracing sharing in an imperative pure calculus. In: *Science of Computer Programming* 172 (2019). Extended version, *CoRR*, https://arxiv.org/abs/1803.05838, pp. 180–202. DOI: 10.1016/j.scico.2018.11.007. Cited on pp. 90, 111.

[33] Paola Giannini, Marco Servetto, Elena Zucca, and James Cone. Flexible recovery of uniqueness and immutability. In: *Theoretical Computer Science* 764 (2019). Extended version, *CoRR*, https://arxiv.org/abs/1807.00137, pp. 145–172. DOI: 10.1016/j.tcs.2018.09.001. Cited on pp. 90, 93, 111, 126.

[34] Jean-Yves Girard. Linear Logic. In: *Theoretical Computer Science* 50 (1987), pp. 1–102. DOI: 10.1016/0304-3975(87)90045-4. Cited on pp. 1, 85.

[35] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In: *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 2012*. Edited by Gary T. Leavens and Matthew B. Dwyer. ACM Press, 2012, pp. 21–40. Cited on pp. 124–127, 132.

[36] Alexander Grothendieck. Catégories fibrées et descente. In: *Revêtements étales et groupe fondamental*. Springer, 1971, pp. 145–194. Cited on p. 44.

[37] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In: *European Conference on Object-Oriented Programming, ECOOP 2010*. Edited by Theo D'Hondt. Vol. 6183. Lecture Notes in Computer Science. Springer, 2010, pp. 354–378. Cited on p. 132.

[38] Peter Hanukaev and Harley Eades III. Combining Dependency, Grades, and Adjoint Logic. In: *ACM SIGPLAN International Workshop on Type-Driven Development, TyDe 2023*. Edited by Youyou Cong and Pierre-Évariste Dagand. ACM Press, 2023, pp. 58–70. DOI: 10.1145/3609027.3609408. Cited on p. 130.

[39] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In: *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*. ACM Press, 1999, pp. 132–146. DOI: 10.1145/320384.320395. Cited on pp. 15, 16.

[40] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the rust programming language. In: *Proceedings of ACM on Programming Languages* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154. Cited on p. 132.

[41] Jim Laird, Giulio Manzonetto, Guy McCusker, and Michele Pagani. Weighted Relational Models of Typed Lambda-Calculi. In: *IEEE Symposium on Logic in Computer Science, LICS 2013*. IEEE Computer Society, 2013, pp. 301–310. DOI: 10.1109/LICS.2013.36. Cited on p. 131.

[42] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. In: *Information and Computation* 207.2 (2009), pp. 284–304. DOI: 10.1016/j.ic.2007.12.004. Cited on pp. 59, 76, 78.

[43] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. In: *Information and Computation* 185.2 (2003), pp. 182–210. DOI: 10.1016/S0890-5401(03)00088-9. Cited on pp. 61, 63.

[44] Saunders Mac Lane. *Categories for the working mathematician*. Vol. 5. Springer Science & Business Media, 2013. Cited on p. 43.

[45] Daniel Marshall, Michael Vollmer, and Dominic Orchard. Linearity and Uniqueness: An Entente Cordiale. In: *European Symposium on Programming, ESOP 2022*. Edited by Ilya Sergey. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 346–375. DOI: 10.1007/978-3-030-99336-8_13. Cited on pp. 132, 137.

[46] Conor McBride. 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*. Edited by Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella. Vol. 9600. Lecture Notes in Computer Science. Springer, 2016, pp. 207–233. DOI: 10.1007/978-3-319-30936-1_12. Cited on pp. 7, 10, 14, 89, 129–131.

[47] Mae Milano, Joshua Turcotti, and Andrew C. Myers. A flexible type system for fearless concurrency. In: *Programming Language Design and Implementation, PLDI 2022*. Edited by Ranjit Jhala and Isil Dillig. ACM Press, 2022, pp. 458–473. DOI: 10.1145/3519939.3523443. Cited on p. 132.

[48] Robin Milner. *Functions as processes*. Research Report RR-1154. INRIA, 1990. URL: https://inria.hal.science/inria-00075405. Cited on p. 131.

[49] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. In: *Proceedings of ACM on Programming Languages* 3.ICFP (2019), 110:1–110:30. DOI: 10.1145/3341714. Cited on pp. 7, 10, 15, 40, 129, 130, 136.

[50] Tomas Petricek. *Context-aware programming languages*. PhD thesis. University of Cambridge, 2017. Cited on pp. 129, 136.

[51] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In: *ACM International Conference on Functional Programming, ICFP 2014*. Edited by Johan Jeuring and Manuel M. T. Chakravarty. ACM Press, 2014, pp. 123–135. DOI: 10.1145/2628136.2628160. Cited on pp. 129, 136.

[52] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: Unified Static Analysis of Context-Dependence. In: *Automata, Languages and Programming, ICALP 2013*. Edited by Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg. Vol. 7966. Lecture Notes in Computer Science. Springer, 2013, pp. 385–397. DOI: 10.1007/978-3-642-39212-2_35. Cited on pp. 89, 129, 136.

[53] Emily Riehl. *Category theory in context*. Courier Dover Publications, 2017. Cited on p. 43.

[54] Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. Effects and Coeffects in Call-By-Push-Value (Extended Version). In: *CoRR* abs/2311.11795 (2023). DOI: 10.48550/ARXIV.2311.11795. URL: https://doi.org/10.48550/arXiv.2311.11795. Cited on pp. 16, 129.

[55] Guannan Wei, Oliver Bracevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. In: *CoRR* abs/2307.13844 (2023). DOI: 10.48550/ARXIV.2307.13844. URL: https://doi.org/10.48550/arXiv.2307.13844. Cited on p. 133.

[56] James Wood and Robert Atkey. A Framework for Substructural Type Systems. In: *European Symposium on Programming, ESOP 2022*. Edited by Ilya Sergey. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 376–402. DOI: 10.1007/978-3-030-99336-8_14. Cited on pp. 7, 10, 14, 89, 131.

[57] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. In: *Information and Computation* 115.1 (1994), pp. 38–94. DOI: 10.1006/inco.1994.1093. Cited on p. 79.