A Core Quantitative Coeffect Calculus

Aloïs Brunel[1], Marco Gaboardi[2], Damiano Mazza[1], and Steve Zdancewic[3]

[1] CNRS, UMR 7030, LIPN, Université Paris 13, Sorbonne Paris Cité
[2] University of Dundee
[3] University of Pennsylvania

Abstract. Linear logic is well known for its resource-awareness, which hasvinspired the design of several resource management mechanisms in programming language design. Its resource-awareness arises from the distinction between linear, single-use data and non-linear, reusable data. The latter is marked by the so-called exponential modality, which, from the categorical viewpoint, is a (monoidal) comonad.
Monadic notions of computation are well-established mechanisms used to express effects in pure functional languages. Less well-established is the notion of comonadic computation. However, recent works have shown the usefulness of comonads to structure context dependent computations. In this work, we present a language $\ell\mathcal{R}\mathsf{PCF}$ inspired by a generalized interpretation of the exponential modality. In $\ell\mathcal{R}\mathsf{PCF}$ the exponential modality carries a label—an element of a semiring $\mathcal{R}$—that provides additional information on how a program uses its context. This additional structure is used to express comonadic type analysis.

1 Introduction

Linear Types. The ideas of linear logic [17] have found several applications in programming languages. The most popular aspect of linear logic is certainly the *distinction* it makes between objects that can be used exactly once and objects that can be used several times—zero or more. This distinction allows type systems to introduce the concept of *usage* that can be exploited to reason about *resources* in various contexts such as explicit memory management, complexity analysis or process specification. The explicit manipulation of resources is obtained formally by introducing the so-called *exponential modality* ! that distinguishes two kinds of types *linear* $A, B, \ldots$—for objects that can be used only once—and *non-linear* $!A, !B, \ldots$—for objects that can be used several times.

Monads and effects. The use of monads in programming languages, originally introduced in category theory, was pioneered by Moggi [28] as a way to structure the semantics of his computational lambda calculus. The use of monads was further advocated by Wadler [39] and they have since found important applications in the development of Haskell. A monad T represents a notion of *computation from a value* that is obtained by distinguishing *value types* $A, B, \ldots$—the types of the language values—from *computation types* $TA, TB, \ldots$—the types of computations over values. From a different perspective a monadic type TA can be seen as a computation that outputs a value of type A and that produces the *effect* described by T on its environment—e.g. a change in the state, input-output operations, *etc. Effect systems* [35] were developed independently as a way to use static type analysis to understand how programs *influence* their environment. Besides this superficial correspondence, monads and effect systems are indeed intimately related as shown by Wadler [40].

Comonads and coeffects. A comonad is the categorical dual of a monad. While monads have grown very popular in the recent years, comonads are still not well-known and probably less understood. Recent works [37, 38, 32] have proposed to interpret a comonad D as a notion of *value in a context* that is obtained by distinguishing *value types* $A, B, \ldots$—the types of the language values—from *contextual types* $DA, DB, \ldots$—the types of values in contexts. From a different perspective, a comonadic type DA can be seen as a computation that consumes a value of type A producing the *coeffect* described by D on its environment. The coeffect described by D can be seen as a requirement of the program with respect to the environment—*e.g.* the availability of a resource, a specific prerequisite on the input, *etc.* A general theory of *coeffect systems* has not yet been established, but some steps in this direction have been recently proposed [32].

Our contributions. It is not difficult to see a common pattern here. Indeed, linear types, monads and comonads are all ways to structure computations by interpreting types in two different ways. The correspondence clearly goes farther: as we already said, monad and comonad are dual notions; moreover, it is a well-known fact that the linear logic exponential modality ! has the structure of a (monoidal) comonad [26]; finally, both the computational λ-calculus and the linear λ-calculus can be embedded in the adjoint calculus, where their relations is revealed through an adjunction [6].

Besides these technical similarities, one may wonder if the comonadic structure of the modality ! can also be used to design general comonadic analyses.

The answer we give in the present work is affirmative. Starting from linear logic, we derive a core PCF-like comonadic language, named $\ell\mathcal{R}$PCF, that is able to express at the same time quantitative reasoning and general comonadic reasoning. Our proposal follows a simple remark (which will not surprise linear logic semantics experts): in many concrete models of linear logic several different interpretations of the ! modality (in terms of abstract resources) are possible. In the present work, we make explicit in the syntax these different interpretations by introducing a modality $!_r$ indexed by an element r of an abstract structure $\mathcal{R}$—a *structural semiring*—that naturally arises from the structural rules of linear logic. Interestingly, this abstract structure also permits general coeffect analyses like the ones previously studied by Petricek *et al.* [32]

A key ingredient of $\ell\mathcal{R}$PCF is the presence of explicit *co-handlers* coeff, which are typed primitives performing an action $r \in \mathcal{R}$ on the context. For instance, a first example of a simple co-handler in the context of clocked dataflow programming [37, 38, 32], is the primitive next that shifts the clock of the input signal s one step forward. This operation can be seen as a co-handler requiring that the context be able to provide signal information one step in the future. To express this requirement in our framework one can add a co-handler next with the associated action 1 on the context. In fact, $\ell\mathcal{R}$PCF is parametrized over the structure $\mathcal{R}$ and the set of co-handlers. Hence, different comonadic analyses may be performed using different primitives and different structures $\mathcal{R}$.

The focus of our work is on providing a framework for specifying sound type analyses even in the presence of operations—like co-handlers—that change the semantics of the language in terms of contextual operations. For this reason, we instrument the language and the operational semantics—presented in the form of an abstract machine—with informations about the *observable* actions of coeffects. This is achieved by adding to the language an explicit observation $(\!|-|\!)$ that mark where to monitor the behavior of a specific part of the program during the computation. We prove a parametric soundness theorem for the type system with respect to the information collected by the instrumented operational semantics. That is, we associate to each term (through its type derivation) a value in $\mathcal{R}$ and we show that this value approximates the information that can be observed at runtime with the instrumented operational semantics. This result is proved by defining a general quantitative realizability model, based on biorthogonality and parametrized over $\mathcal{R}$. Finally, we sketch a denotational semantics for $\ell\mathcal{R}$PCF in the form of an interpretation of $\ell\mathcal{R}$PCF programs in a categorical structure describing the properties of $\mathcal{R}$.

Summarizing, our contributions are:

– A quantitative comonadic core language inspired by linear logic semantics. This language is parametrized over an abstract structure $\mathcal{R}$ and over a set of coeffect primitives (co-handlers). By instantiating $\mathcal{R}$ with concrete structures and by choosing particular sets of co-handlers we are able to perform several context-dependent analysis.
– A parametrized quantitative realizability technique used to prove the soundness of the different analyses. The realizability is parametrized over $\mathcal{R}$ and the set of coeffect primitives. The soundness is proved with respect to an abstract machine reduction relation.
– The description of a categorical model, based on Melliès's work on parametric comonads [27**?**], showing the abstract structure needed to interpret our language. Such a categorical semantics provides a base for a comparison with usual semantics of linear logic.

2 The $\ell\mathcal{R}$PCF Language

Syntax: linear constructors and coeffects. The language of $\ell\mathcal{R}$PCF, defined in Fig. 1.a, is a linearized version of PCF (*i.e.*, with explicit constructors for the modalities of linear logic) extended with coeffects.

The !-constructor and `let` bindings are standard in languages designed using the proofs-as-programs correspondence with linear logic. Here, they are used to explicitly track the use of the coeffects in expressions. We consider three kinds of values: numerals $\underline{n}$, abstractions $\lambda x.e$ and expressions of the form $!e$. The latter are useful to delimit the scope of coeffects.

The construction coeff(e) wraps expression e in a *coeffect handler* coeff. Note that coeff is a metavariable, ranging over a (finite) set of *coeffect handler identifiers* or, more simply, *co-handlers*. We leave the set of co-handlers unspecified, as a parameter of $\ell\mathcal{R}$PCF.

$$\begin{array}{rll}
e ::= & x \mid \lambda x.e \mid e\,e \mid \mathsf{let}\ !x = e\ \mathsf{in}\ e \mid !e \mid (\!| e |\!) \mid \mathsf{coeff}(e) \mid & \text{(expressions)} \\
& \underline{n} \mid \mathsf{s}(e) \mid \mathsf{case}\ e\ \mathsf{of}\ \underline{0} \to e\ \mathsf{else}\ \underline{x+1} \to e \mid \mathsf{fix}\ x.e & \\
v ::= & \lambda x.e \mid \underline{n} \mid !e & \text{(values)} \\
\hline
A ::= & \mathsf{Nat} \mid !_r A \mid A \multimap A \qquad r \in \mathcal{R} & \text{(types)} \\
\Gamma ::= & \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]_r \qquad r \in \mathcal{R} & \text{(contexts)} \\
\hline
c ::= & (e, \rho) & \text{(closures)} \\
\rho ::= & [] \mid \rho \cdot [x/c] & \text{(environments)} \\
\pi ::= & \diamond \mid \langle c \rangle.\pi \mid \langle x, e, \rho \rangle.\pi \mid \langle \mathsf{coeff} \rangle.\pi \mid \langle \mathsf{s} \rangle.\pi \mid \langle e_1, x, e_2, \rho \rangle.\pi & \text{(stacks)} \\
C ::= & (c, \pi) & \text{(configurations)}
\end{array}$$

Fig. 1. $\ell\mathcal{R}$PCF (a) grammar (b) typing (c) abstract machine configurations

Finally, the construct $(\!| - |\!)$ is an *observation*. It has no computational value but makes our quantitative analysis more flexible. By arbitrarily introducing observations in expressions, we can track the behavior of specific subterms during a computation, giving more power to our quantitative soundness result.

Structural semirings. The other main parameter of $\ell\mathcal{R}$PCF (or, rather, of its type system) is the following algebraic structure.

Definition 1 (Structural semiring) *A* structural semiring*, denoted in general by $\mathcal{R}$, is a tuple $(\mathcal{R}, +, \mathbf{0}, \star, \mathbf{1}, \preceq)$ such that:*

- *$(\mathcal{R}, +, \mathbf{0}, \star, \mathbf{1})$ is a unit semiring, that is:*
  - *$(\mathcal{R}, +, \mathbf{0})$ is a commutative monoid;*
  - *$(\mathcal{R}, \star, \mathbf{1})$ is a monoid;*
  - *multiplication distributes over addition, i.e., for all $p, q, r \in \mathcal{R}$:*
    - *$r \star (p + q) = r \star p + r \star q$,*
    - *$(p + q) \star r = p \star r + q \star r$;*
  - *$\mathbf{0}$ is absorbing for multiplication: $p \star \mathbf{0} = \mathbf{0} \star p = \mathbf{0}$ for all $p \in \mathcal{R}$.*
- *$(\mathcal{R}, \preceq)$ is a bounded sup-semilattice, that is:*
  - *$\preceq$ is a partial order on $\mathcal{R}$ such that the least upper bound of every two elements $p, q \in \mathcal{R}$ exists and is denoted by $p \vee q$;*
  - *there is a least and greatest element, the latter being denoted by ∞.*

Moreover, the following compatibility conditions hold, for all $p, q, r \in \mathcal{R}$:

- *$\mathbf{0}$ is the least element;*
- *$p \preceq q$ implies $p + r \preceq q + r$, $r \star p \preceq r \star q$ and $p \star r \preceq q \star r$.*

Note that the compatibility conditions imply that ∞ is absorbing for addition and that it is idempotent w.r.t. both operations.

The notion of structural semiring arises naturally from the structural rules of linear logic, hence the name.[4] It is possible to give a categorical generalization of this structure, which is used for describing the denotational models of $\ell\mathcal{R}$PCF (*cf.* Sect. 4).

The presence of least upper bounds is not strictly necessary; it is useful to provide a more precise typing of the `case` construction. Similarly, the existence of a greatest element is postulated only to ensure that fixpoints may be given at least a trivial type.

The following are some notable examples of structural semirings:

- the extended natural numbers $\overline{\mathbb{N}} := \mathbb{N} \cup \{\infty\}$ (usual operations and order);
- the tropical semiring $\mathcal{T} := (\overline{\mathbb{N}}, \min, \infty, +, 0, \geq)$ (note the reversed ordering);
- the arctic semiring $\mathcal{A} := (\overline{\mathbb{N}} \cup \{-\infty\}, \max, -\infty, +, 0, \leq)$;
- the Boolean lattice $\{0, 1\}$, as well as any bounded distributive lattice;
- the probability semiring $\overline{\mathbb{R}}^+$ of non-negative real numbers plus infinity, with the usual operations and order.

Type system. As mentioned above, the type system of $\ell\mathcal{R}$PCF is parametrized over a structural semiring $\mathcal{R}$. Elements of $\mathcal{R}$ can appear in types (defined in Figure Fig. 1.b) as decorations of the exponential modality, as well as in *discharged types* (of the form $[A]_r$) in typing contexts (also defined in Fig. 1.b). Discharged types are not themselves types; they can appear only in contexts and they cannot be nested.

Each co-handler coeff comes with three pieces of information: its *source type* A_{coeff}, its *target type* B_{coeff} and its *coeffect* $r_{\mathsf{coeff}} \in \mathcal{R}$. It also comes with a *coeffect map* φ_{coeff}, which assigns to every value of type A_{coeff} a value of type B_{coeff}. The coeffect map will be required to satisfy a semantic soundness property, which we will give in Sect. 3. We use the term "map" instead of "function" because we do not want to restrict the kind of transitions (of the abstract machine, to be introduced below) we can consider. For instance, φ_{coeff} may be probabilistic or non-deterministic. We consider only unary co-handlers; n-ary co-handlers could be obtained by combining unary co-handlers with the usual tensor product of linear logic, which we do not include here for brevity.

A typing context Γ is a set of typed variables that are either of the form $x : A$ (*linear* variables) or $x : [A]_r$ (*discharged* variables). Discharged variables are a technical artifact useful to implicitly manage variables in contexts. More specifically, if we denote by $[\Gamma]$ a discharged context (a context containing only discharged variables) we can extend the operation $+$ of the semiring to contexts:

$$\begin{aligned}
\emptyset + \Delta &= \Delta \\
(x\!:\![A]_p, \Gamma) + (x\!:\![A]_q, \Delta) &= x\!:\![A]_{p+q}, (\Gamma + \Delta) \\
(x\!:\![A]_p, \Gamma) + \Delta &= x\!:\![A]_p, (\Gamma + \Delta) \qquad \text{if } x \notin \Delta \\
(x\!:\!A, \Gamma) + \Delta &= x\!:\!A, (\Gamma + \Delta) \qquad \text{if } x \notin \Delta
\end{aligned}$$

[4] For the acquainted reader: contraction is addition, with weakening being its neutral element; multiplication comes from crossing the context of a promotion rule, with dereliction being its unit.

$$\frac{}{A <: A}\,\textsf{O-I} \qquad \frac{A <: B \quad q \preceq p}{!_p A <: !_q B}\,\textsf{O-B} \qquad \frac{A' <: A \quad B <: B'}{A \multimap B <: A' \multimap B'}\,\textsf{O-L}$$

$$\frac{A <: B \quad q \preceq p}{[A]_p <: [B]_q}\,\textsf{O-D} \qquad \frac{}{\Gamma <: \Gamma}\,\textsf{O-IC} \qquad \frac{\Gamma <: \Delta \quad A <: B}{\Gamma, x : B <: \Delta, x : A}\,\textsf{O-C}$$

$$\frac{}{x{:}A \vdash x : A}\,\textsf{id} \qquad \frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.e : A \multimap B}\,\textsf{lam} \qquad \frac{\Gamma \vdash e : A \multimap B \qquad \Delta \vdash e' : A}{\Gamma + \Delta \vdash e\, e' : B}\,\textsf{app}$$

$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma, x{:}[A]_{\mathbf{1}} \vdash e : B}\,\textsf{der} \qquad \frac{[\Gamma] \vdash e : B}{r \star [\Gamma] \vdash !e : !_r B}\,\textsf{pr} \qquad \frac{\Gamma \vdash e : !_r A \qquad \Delta, x{:}[A]_r \vdash e' : B}{\Gamma + \Delta \vdash \textsf{let } !x = e \textsf{ in } e' : B}\,\textsf{let}$$

$$\frac{}{\vdash \underline{n} : \textsf{Nat}}\,\textsf{nat} \qquad \frac{\Gamma \vdash e : \textsf{Nat}}{\Gamma \vdash \textsf{s}(e) : \textsf{Nat}}\,\textsf{succ} \qquad \frac{[\Gamma], x{:}[A]_p \vdash e : A \qquad \mathbf{1} + p \star q \preceq q}{q \star [\Gamma] \vdash \textsf{fix } x.e : A}\,\textsf{fix}$$

$$\frac{\Gamma \vdash e : \textsf{Nat} \qquad \Delta \vdash e_1 : A \qquad \Delta, x{:}\textsf{Nat} \vdash e_2 : A}{\Gamma + \Delta \vdash \textsf{case } e \textsf{ of } \underline{0} \to e_1 \textsf{ else } \underline{x+1} \to e_2 : A}\,\textsf{case}$$

$$\frac{\Delta \vdash e : B \qquad \Gamma <: \Delta}{\Gamma, \Xi \vdash e : B}\,\textsf{sub} \qquad \frac{[\Gamma] \vdash e : A_{\textsf{coeff}}}{r_{\textsf{coeff}} \star [\Gamma] \vdash \textsf{coeff}(e) : B_{\textsf{coeff}}}\,\textsf{coeff}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \llparenthesis e \rrparenthesis : A}\,\textsf{obs}$$

Fig. 2. (a) Subtyping rules (b) Typing rules

Note that the addition of contexts is partial: $\Gamma + \Delta$ is defined only if Γ and Δ do not share any linear variable declaration. In what follows, the use of this operation implicitly means that this condition is met. Similarly, the action of an element $r \in \mathcal{R}$ on a discharged context $[\Gamma]$, denoted by $r \star [\Gamma]$, is defined by induction on the size of $[\Gamma]$ as: $r \star \emptyset = \emptyset$ and $r \star (x{:}[A]_p, [\Gamma]) = x{:}[A]_{r \star p}, r \star [\Gamma]$. We also extend to contexts the partial order of $\mathcal{R}$, by introducing *subtyping* between types. A subtyping judgment $A <: B$ can be obtained using the rules in Fig. 2.a. The subtyping rules are rather standard with the exception of rules O-B and O-D, which lift the semiring partial order to types; notice that these rules are contravariant in the elements of the semiring.

As usual, typing judgments are of the form $\Gamma \vdash e : A$, where in our case Γ may contain both linear and discharged variables. The typing rule are in Fig. 2.b.

The rule der introduces a discharged variable starting from a linear variable. This rule may be seen as a quantitative analog of the *dereliction* principle of

x	$\rho \cdot [x/(e,\rho')]$	π	$\rightarrow_{\mathsf{v}}$	e	ρ'	π
$\lambda x.e$	ρ	$\langle c\rangle.\pi$	$\rightarrow_{\lambda}$	e	$\rho \cdot [x/c]$	π
$e_1\, e_2$	ρ	π	$\rightarrow_{@}$	e_1	ρ	$\langle(e_2,\rho)\rangle.\pi$
$\mathsf{let}\ !x = e_1\ \mathsf{in}\ e_2$	ρ	π	$\rightarrow_{\mathsf{1}}$	e_1	ρ	$\langle x, e_2, \rho\rangle.\pi$
$!e_1$	ρ_1	$\langle x, e_2, \rho_2\rangle.\pi$	$\rightarrow_{!}$	e_2	$\rho_2 \cdot [x/(e_1,\rho_1)]$	π
$\mathsf{s}(e)$	ρ	π	$\rightarrow_{\mathsf{s}}$	e	ρ	$\langle \mathsf{s}\rangle.\pi$
$\underline{n}$	ρ	$\langle \mathsf{s}\rangle.\pi$	$\rightarrow_{+}$	$\underline{n+1}$	ρ	π
$\left(\begin{array}{c}\mathsf{case}\ e\ \mathsf{of}\ \underline{0} \rightarrow e_1\\ \mathsf{else}\ \underline{x+1} \rightarrow e_2\end{array}\right)$	ρ	π	$\rightarrow_{\mathsf{i}}$	e	ρ	$\langle e_1, x, e_2, \rho\rangle.\pi$
$\underline{0}$	ρ_1	$\langle e_1, x, e_2, \rho_2\rangle.\pi$	$\rightarrow_{\mathsf{z}}$	e_1	ρ_2	π
$\underline{n+1}$	ρ_1	$\langle e_1, x, e_2, \rho_2\rangle.\pi$	$\rightarrow_{\mathsf{e}}$	e_2	$\rho_2 \cdot [x/(\underline{n},\rho_1)]$	π
$\mathsf{fix}\ x.e$	ρ	π	$\rightarrow_{\mathsf{f}}$	e	$\rho \cdot [x/(\mathsf{fix}\ x.e, \rho)]$	π
$\mathsf{coeff}(e)$	ρ	π	$\rightarrow_{\mathsf{c}}$	e	ρ	$\langle \mathsf{coeff}\rangle.\pi$
v	ρ	$\langle \mathsf{coeff}\rangle.\pi$	$\rightarrow_{\mathsf{x}}$	$\varphi_{\mathsf{coeff}}(v)$	ρ	π
$\llparenthesis e \rrparenthesis$	ρ	π	$\rightarrow_{\mathsf{o}}$	e	ρ	π

Fig. 3. The $\mathcal{K}^{\mathcal{R}}$ machine.

linear logic: $!A \multimap A$. A way of reading this rule is: "a variable with coeffect $\mathbf{1}$ is also linear". Similarly, the pr rule corresponds to a quantitative version of the *promotion* rule of linear logic. Note that this rule is in fact a scheme for rules parametrized by an element $r \in \mathcal{R}$. A way of reading this rule is: "if a co-handler whose coeffect is r is to operate on an expression e, then r has to act on the context of e". The rule sub is at the same time the rule for the subtyping and for *weakening* (of the context Ξ); indeed, the system is actually *affine*, not strictly linear. The rule let is responsible for removing discharged variables. This can be seen as an analog (or dual) of the `let` of the computational λ-calculus. Note that this rule, as well as all the binary rules, uses the operation $+$ to merge the contexts of the two premises. This is because the resulting coeffect is the sum of the coeffects in the two premises.

The rule coeff is the rule for typing co-handler expressions. It is parametrized on the particular co-handler. We could have chosen to have this rule as derivable from an application of the pr rule—introducing an extra !-operator in the expression—and an application of an axiom rule introducing the co-handler. We prefer this formulation so that co-handlers are always applied in expressions—and we also avoid the use of an extra !-operator.

The additive management of the context Δ in the two branches of the case rule is standard for languages inspired by linear logic. Note that the existence of least upper bounds is useful here for type-inference: it allows to find minimal discharged types to build the context Δ. The last rule deserving some explanation is fix. This is parametrized by an element $q \in \mathcal{R}$ that has to satisfy the side condition $\mathbf{1} + q \star p \preceq q$. Note that for every $p \in \mathcal{R}$ the element ∞ satisfies this condition. However, in general there may be other elements satisfying it.

The abstract machine. The operational semantics we consider is provided by an adaptation of the Krivine abstract machine [22]. In particular, we extend (in a standard way) Krivine's machine to deal with natural numbers, conditional and fixpoint. The basic components of the machine (closures, environments, stacks, configurations) are defined in Fig. 1.c. Stacks are also assigned a *weight*:

Definition 2 (Weight of a stack) *Let π be a stack. Its* weight $w(\pi)$ *is the element of $\mathcal{R}$ defined by induction on π as follows:*

- $w(\diamond) = \mathbf{1}$*;*
- $w(\langle \mathsf{coeff} \rangle.\pi') = w(\pi') \star r_{\mathsf{coeff}}$*;*
- $w(\kappa.\pi') = w(\pi')$ *in all other cases.*

A *state* of the machine is a pair (C, r), where C is a configuration and $r \in \mathcal{R}$. This latter, called the *observable quantity*, must be seen as the value of a counter. It adds a quantitative aspect to the operational semantics of $\ell\mathcal{R}\mathsf{PCF}$.

The transitions of the $\mathcal{K}^{\mathcal{R}}$ machine are given in Fig. 3. The counter of the machine is left untouched by all transitions except the o transition: if $C = (\llparenthesis e \rrparenthesis, \rho, \pi)$ and $C' = (e, \rho, \pi)$, then the state (C, r) evolves to $(C', r + w(\pi))$. We write $C \to C'$ when we do not want to specify the kind of transition.

In general, we are interested in computations of the shape $((e, [], \diamond), \mathbf{0}) \to^* ((v, \rho, \diamond), r)$, *i.e.*, computations that evaluate expressions in the empty environment and the empty stack starting with an observable quantity of $\mathbf{0}$. In this case, we can say that r is the *observable quantity* of the computation. The goal of our type analysis is to provide by static analysis a bound to this quantity. This is obtained by a quantitative realizability technique that we present in the next section.

3 Quantitative Realizability

This section presents the construction of a realizability interpretation suitable for modeling $\ell\mathcal{R}\mathsf{PCF}$ as parameterized by an arbitrary structural semiring $\mathcal{R}$. However, to soundly handle the fixpoint typing rule, it is necessary to "step index" the construction. Fortunately, such step indexing can itself be smoothly added using a structural semiring.

In the rest of the section we fix an arbitrary structural semiring $\mathcal{R}$ and we consider the structural semiring $\mathcal{R} \oplus \mathcal{T}$, where $\mathcal{T}$ is the tropical semiring defined in Sect. 2. The elements of $\mathcal{R} \oplus \mathcal{T}$, which we denote by α, β, γ, are pairs of the form (p, m) where $p \in \mathcal{R}$ and $m \in \overline{\mathbb{N}}$. The operations and order relation on these elements are (abusively) denoted like the operations and order relation of $\mathcal{R}$: $(p, m) + (q, n) = (p + q, \min(m, n))$ with neutral element $(\mathbf{0}, \infty)$, $(p, m) \star (q, n) = (p \star q, m + n)$ with neutral element $(\mathbf{1}, 0)$, and $(p, m) \preceq (q, n)$ iff $p \preceq q$ and $n \leq m$ (note the reverse ordering on integers).

The elements of $\mathcal{R}$ may be (monotonically) embedded in $\mathcal{R} \oplus \mathcal{T}$ through the additive endomorphism $p \mapsto (p, \infty)$ and the multiplicative endomorphism $p \mapsto (p, 0)$. In the sequel, we tacitly apply such embeddings to treat elements of $\mathcal{R}$ as elements of $\mathcal{R} \oplus \mathcal{T}$, using the suitable endomorphism according to the operation of interest. For instance, given a stack π, we write $\alpha + w(\pi)$ to actually mean $\alpha + (w(\pi), \infty)$, and we write $w(\pi) \star \alpha$ to actually mean $(w(\pi), 0) \star \alpha$.

Orthogonality. In what follows, we associate with each transition $C \to C'$ of the $\mathcal{K}^{\mathcal{R}}$ machine a function $\theta[C \to C'] : \mathcal{R} \oplus \mathcal{T} \to \mathcal{R} \oplus \mathcal{T}$ which is the identity in all cases except:

- when $C \to_{\mathsf{f}} C'$, in which case we set $\theta[C \to C'](p,m) = (p, m+1)$;
- when $C = (\llparenthesis e \rrparenthesis, \rho, \pi)$ and $C \to_{\mathsf{o}} C'$, in which case we set $\theta[C \to C'](p,m) = (p + w(\pi), m)$.

Definition 3 (Pole) *A* pole *is a family $\perp\!\!\!\perp = (\perp\!\!\!\perp_\alpha)_{\alpha \in \mathcal{R} \oplus \mathcal{T}}$ of sets of configurations such that:*

- Saturation: *if $C' \in \perp\!\!\!\perp_\alpha$ and $C \to C'$, then $C \in \perp\!\!\!\perp_{\theta[C \to C'](\alpha)}$;*
- Monotonicity: *$\alpha \preceq \beta$ implies $\perp\!\!\!\perp_\alpha \subseteq \perp\!\!\!\perp_\beta$;*
- Approximation: *for all $p \in \mathcal{R}$, $\perp\!\!\!\perp_{(p,0)}$ is the set of all configurations and $\bigcap_{n \in \mathbb{N}} \perp\!\!\!\perp_{(p,n)} = \perp\!\!\!\perp_{(p,\infty)}$;*
- Weakening: *for all α and $(e, \rho, \pi) \in \perp\!\!\!\perp_\alpha$, if $y_1, \dots, y_k$ do not appear free in e, then, for all closures $c_1, \dots, c_k$, $(e, \rho \cdot [y_1/c_1] \cdots [y_k/c_k], \pi) \in \perp\!\!\!\perp_\alpha$.*

Definition 4 (Weighted closures and stacks, orthogonality) *A* weighted closure *(resp.* weighted stack*) is a pair (c, α) (resp. (π, α)) where c is a closure (resp. π is a stack) and $\alpha \in \mathcal{R} \oplus \mathcal{T}$.*

Let $((e, \rho), \alpha)$, (π, β) be a weighted closure and stack, respectively, and let $\perp\!\!\!\perp$ be a pole. We define the orthogonality relation *w.r.t. $\perp\!\!\!\perp$ by*

$$((e, \rho), \alpha) \perp (\pi, \beta) \quad \textit{iff} \quad (e, \rho, \pi) \in \perp\!\!\!\perp_{w(\pi) \star \alpha + \beta}.$$

Intuitively, the pole expresses a notion of correctness, and ortogonality means that the closure (program) and stack (environment) interact correctly. In Sect. 5 we will give explicit examples of poles and clarify this intuition.

The orthogonality relation lifts to sets of weighted closures X and sets of weighted stacks Y as usual: $X^\perp := \{\ (\pi, \beta) \mid \forall (c, \alpha) \in X, (c, \alpha) \perp (\pi, \beta)\ \}$, and $Y^\perp := \{\ (c, \alpha) \mid \forall (\pi, \beta) \in Y, (c, \alpha) \perp (\pi, \beta)\ \}$. The *biorthogonality operator* $(.)^{\perp\perp}$ on sets of weighted closures is then a closure operator.

Lemma 5 *Suppose that X is a set of weighted closures or weighted stacks. Then: (i) $X \subseteq X^{\perp\perp}$; (ii) $Y \subseteq X$ implies $X^\perp \subseteq Y^\perp$; (iii) $X^{\perp\perp\perp} = X^\perp$.*

Moreover, it is easy to see that the properties of the pole are transferred to biorthogonally-closed sets of weighted closures:

Lemma 6 *Let X be a set of weighted closures. Then:*

1. *if, for all $n \in \mathbb{N}$, $(c, (p, n)) \in X$, then $(c, (p, \infty)) \in X^{\perp\perp}$;*
2. *if $((e, \rho), \alpha) \in X$, $\alpha \preceq \beta$ and if $y_1, \dots, y_k$ are variables not appearing free in e, then $((e, \rho \cdot [y_1/c_1] \cdots [y_k/c_k]), \beta) \in X^{\perp\perp}$ for all closures $c_1, \dots, c_k$.*

Interpretation. We are now going to assign to each type a set of weighted closures. We first define in Fig. 4 two operations $\multimap$ and $!_r$, along with the set Nat (for convenience, we use the same notation as the type).

Definition 7 (Interpretation, realizability, adaptation) *Let A be a type. Its* interpretation *$\|A\|$ is the set of weighted closures defined as follows:*

$$\|\mathsf{Nat}\| := \mathsf{Nat}^{\perp\perp} \qquad \|A \multimap B\| := (\|A\| \multimap \|B\|)^{\perp\perp} \qquad \|!_r A\| := (!_r \|A\|)^{\perp\perp}$$

The realizability relation *$(c, \alpha) \Vdash A$ is valid if and only if $(c, \alpha) \in \|A\|$.*

Note that realizability depends on the pole. We say that a pole is adapted *if we have $(\diamond, (\mathbf{0}, \infty)) \in \|A\|^\perp$ for every type A.*

$$\begin{aligned}
\mathsf{Nat} &:= \{\ ((\underline{n},[]),(\mathbf{0},\infty)) \mid n \in \mathbb{N}\ \} \\
X \multimap Y &:= \{\ ((\lambda x.e,\rho),\alpha) \mid \forall (c',\beta) \in X,\ ((e,\rho\cdot[x/c']),\alpha+\beta) \in Y^{\perp\perp}\ \} \\
r \star X &:= \{\ (c,(r\star p,m)) \mid (c,(p,m)) \in X\ \} \\
!_r X &:= \{\ ((!e,\rho),\alpha) \mid ((e,\rho),\alpha) \in r \star X\ \}
\end{aligned}$$

Fig. 4. Realizability operations. X and Y are generic sets of weighted closures.

Soundness. We start by introducing the notions needed to state the soundness theorem. We first need to extend the realizability relation to open terms. Then, we will define what it means for a typing judgment and a typing rule to be sound.

Definition 8 (Sound environment) *Suppose $\boldsymbol{\gamma}$ is a sequence $\gamma_1,\ldots,\gamma_n$ of elements of $\mathcal{R}\oplus\mathcal{T}$. We say that an environment $\rho = [x_1/c_1]\cdots[x_n/c_n]$ is $\boldsymbol{\gamma}$-sound with respect to $\Gamma = x_1 : A_1,\ldots,x_k : A_k, x_{k+1} : [A_{k+1}]_{r_{k+1}},\ldots,x_n : [A_n]_{r_n}$, and we write $(\rho,\boldsymbol{\gamma}) \Vdash \Gamma$, if $(c_i,\gamma_i) \in \|A_i\|$ for $1 \le i \le k$, and $(c_i,\gamma_i) \in r_i \star (\|A_i\|)$ for $k+1 \le i \le n$.*

In what follows, if $\boldsymbol{\gamma}$ is a sequence $\gamma_1,\ldots,\gamma_n$ of elements of $\mathcal{R}\oplus\mathcal{T}$, we denote by $\sum\boldsymbol{\gamma}$ the element $\gamma_1+\ldots+\gamma_n$ (which is $(\mathbf{0},\infty)$ if $n=0$).

Definition 9 (Sound judgment and rules) *Let $p \in \mathcal{R}$. We say that the judgment $\Gamma \vdash e : A$ is p-*sound *if $(\rho,\boldsymbol{\gamma}) \Vdash \Gamma$ implies $((e,\rho), p+\sum\boldsymbol{\gamma}) \Vdash A$.*

*Consider a typing rule R whose premises are the judgments $J_1,\ldots,J_n$ and whose conclusion is the judgment K. Let $\phi : \mathcal{R}^n \to \mathcal{R}$. We say that R is ϕ-*sound *if for all $p_1,\ldots,p_n \in \mathcal{R}$ such that J_i is p_i-sound, then K is $\phi(p_1,\ldots,p_n)$-sound.*

Any judgment obtained by composition of sound rules is itself a sound judgment. Hence, to prove soundness of our type system with respect to the realizability semantics, it will suffice to prove the soundness of each typing rule.

If R is an n-ary rule of our type system, we associate to it a *soundness function* of type $\mathcal{R}^n \to \mathcal{R}$ denoted by $\phi[\mathsf{R}]$. The definition is given in Fig. 5, where we use meta-λ-notation. For instance, $\phi[\mathsf{obs}]$ is the function taking an element $p \in \mathcal{R}$ and returning the element $p+\mathbf{1}$ of $\mathcal{R}$.

If δ is a typing derivation of conclusion $\Gamma \vdash e : A$, then we may assign to it a *soundness element* $p[\delta] \in \mathcal{R}$, defined by composing the $\phi[\mathsf{R}]$ for each rule R used in δ, inductively. Then, we have the following:

Theorem 10 (Soundness) *The conclusion of every derivation δ is $p[\delta]$-sound.*

The proof of the above result, which we omit here for space reasons, is conditional to the following hypothesis being verified, for every co-handler coeff, which we call *soundness* of coeff:

$$(\pi,\beta) \in \|B_{\mathsf{coeff}}\|^{\perp} \text{ implies } (\langle\mathsf{coeff}\rangle.\pi\beta) \in \|A_{\mathsf{coeff}}\|^{\perp}.$$

$$\begin{array}{rlrlrl}
\phi[\mathsf{id}] &:= \mathbf{0} & \phi[\mathsf{lam}] &:= \lambda p.p & \phi[\mathsf{app}] &:= \lambda(p,q).p+q \\
\phi[\mathsf{der}] &:= \lambda p.p & \phi[\mathsf{pr}] &:= \lambda p.r \star p & \phi[\mathsf{let}] &:= \lambda(p,q).p+q \\
\phi[\mathsf{nat}] &:= \mathbf{0} & \phi[\mathsf{succ}] &:= \lambda p.p & \phi[\mathsf{fix}] &:= \lambda p.q \star p \\
\phi[\mathsf{sub}] &:= \lambda p.p & \phi[\mathsf{coeff}] &:= \lambda p.r_{\mathsf{coeff}} \star p & \phi[\mathsf{obs}] &:= \lambda p.p+\mathbf{1}
\end{array}$$
$$\phi[\mathsf{case}] := \lambda(p,q,r).p + (q \vee r)$$

Fig. 5. Soundness functions of the typing rules of Fig. 2.b. The arity of each function is the same as that of its associated rule; p, q, r correspond to the first, second and third premises, respectively (from left to right).

This is the semantic condition on coeffect maps which we mentioned when we introduced the type system.

In case the pole is adapted, we obtain the following important result:

Corollary 11 *If $\vdash e : A$ via a typing derivation δ, then $(e, [], \diamond) \in \perp\!\!\!\perp_{(p[\delta],\infty)}$.*

Since $C \in \perp\!\!\!\perp_{(p,\infty)}$ usually means "the configuration C uses at most p resources", we have that, for properties that can be expressed using an adapted pole, typing derivations of $\ell\mathcal{R}\mathsf{PCF}$ imply quantitative bounds on the execution of the typed expression.

4 Categorical Semantics

Our framework has a rich underlying structure that we describe in categorical terms in this section. The first step is introducing bimonoidal categories (formerly called *ring categories* [25]), which are a "categorification" of the notion of semiring. The most synthetic way of defining a bimonoidal category is saying that it is a one-object category enriched over symmetric monoidal categories [19]. Spelled out, this means that a bimonoidal category is a structure $(\mathcal{S}, +, 0, \star, 1, \mathsf{d}^l, \mathsf{d}^r, \mathsf{a}^l, \mathsf{a}^r)$ such that $(\mathcal{S}, +, 0)$ is a symmetric monoidal category, $(\mathcal{S}, \star, 1)$ is a monoidal category and $\mathsf{d}^l, \mathsf{a}^l, \mathsf{d}^r, \mathsf{a}^r$ are structure maps ensuring distibutivity and absorption laws. A certain numer of coherence diagrams are required to commute, of course; the precise definition may be found in [19].

Next, we introduce a notion of parametric comonad, that we take from [27]. In what follows, we will deal with two categories $\mathcal{S}$ and $\mathcal{A}$ and we shall use x, y (resp. a, b) as placeholders for the arguments of a functor of domain $\mathcal{S}$ (resp. $\mathcal{A}$), *e.g.* an endofunctor F of $\mathcal{A}$ will be denoted by $F(a)$, whereas we use p, q, r (resp. A, B) to range over the objects of $\mathcal{S}$ (resp. $\mathcal{A}$).

Definition 12 (Positive action) *A* positive action *of a monoidal category $(\mathcal{S}, \star, 1)$ on a category $\mathcal{A}$ is a functor $\centerdot : \mathcal{S} \times \mathcal{A} \longrightarrow \mathcal{A}$ with two natural transformations $\delta : (x \star y) \centerdot a \Longrightarrow x \centerdot (y \centerdot a)$ and $\varepsilon : 1 \centerdot a \Longrightarrow a$ such that the following* *diagrams commute:*

$$\begin{tikzcd}
(p \star (q \star r)) \centerdot A \arrow[r, "\alpha^\star \centerdot A"] \arrow[d, "\delta"'] & ((p \star q) \star r) \centerdot A \arrow[r, "\delta"] & (p \star q) \centerdot (r \centerdot A) \arrow[d, "\delta"] \\
p \centerdot ((q \star r) \centerdot A) \arrow[rr, "p \centerdot \delta"'] & & p \centerdot (q \centerdot (r \centerdot A))
\end{tikzcd}
\qquad
\begin{tikzcd}
(1 \star p) \centerdot A \arrow[r, "\lambda^\star"] \arrow[d, "\delta"'] & p \centerdot A \arrow[d, equal] & (p \star 1) \centerdot A \arrow[l, "\rho^\star"'] \arrow[d, "\delta"] \\
1 \centerdot (p \centerdot A) \arrow[r, "\varepsilon"'] & p \centerdot A & p \centerdot (1 \centerdot A) \arrow[l, "p \centerdot \varepsilon"]
\end{tikzcd}$$

We now generalize Definition 12 to the case of a bimonoidal category acting on a symmetric monoidal category. This should be seen as a categorification of the "raising to a power" action: the natural transormations required correspond to the usual, elementary laws of exponentiation (such as $A^{p+q} = A^p A^q$, $A^0 = I$, and so on). Although not contained in either [27] or [**?**], the definition was still suggested to the authors by Melliès.

Definition 13 (Exponential action) *Let $(\mathcal{A}, \otimes, I)$ be a symmetric monoidal category, and let $(\mathcal{S}, +, 0, \star, 1)$ be a bimonoidal category. An* exponential action *of $\mathcal{S}$ on $\mathcal{A}$ is a positive action $(\centerdot, \delta, \varepsilon)$ of $(\mathcal{S}, \star, 1)$ on $\mathcal{A}$ together with four natural transformations $\mathsf{c} : (x + y) \centerdot a \Longrightarrow x \centerdot a \otimes y \centerdot a$, $\mathsf{w} : 0 \centerdot a \Longrightarrow I$, $\mathsf{m} : x \centerdot a \otimes x \centerdot b \Longrightarrow x \centerdot (a \otimes b)$, and $\mathsf{n} : I \Longrightarrow x \centerdot I$ such that*

- *for every object A of $\mathcal{A}$, the natural transformations $\mathsf{c}^A, \mathsf{w}^A$ induced by fixing the parameter A in c, w make the functor $x \centerdot A : (\mathcal{S}, +, 0) \to \mathcal{A}$ symmetric comonoidal (i.e. oplax monoidal);*
- *for every object p of $\mathcal{S}$, the natural transformations $\mathsf{m}^p, \mathsf{n}^p$ induced by fixing the parameter p in m, n make the endofunctor $p \centerdot a$ of $\mathcal{A}$ symmetric monoidal.*

Furthermore, we require 12 *diagrams to commute, which are fairly natural but cannot be included for space reasons.*

Definition 14 (Bounded exponential situation) *A* bounded exponential situation *consists of the following data:*

- *a symmetric monoidal closed category $(\mathcal{A}, \otimes, I, \multimap)$;*
- *a bimonoidal category $(\mathcal{S}, +, 0, \star, 1)$ with finite coproducts (not necessarily expressed by $+$) and a terminal object, in which 0 is initial;*
- *an exponential action $!$ of $\mathcal{S}^{\mathrm{op}}$ on $\mathcal{A}$, for which we use the notation $!_p A$ (with p an object of $\mathcal{S}$ and A an object of $\mathcal{A}$).*

A bounded exponential situation is affine *if I is terminal in $\mathcal{A}$.*

An affine bounded exponential situation is enough to interpret the typing rules of $\ell\mathcal{R}\mathsf{PCF}$ (Fig. 2). The category $\mathcal{S}$ is a "category of bounds": it is the generalization of a structural semiring, in which an arrow $p \to q$ may be seen as a proof that $p \preceq q$. In fact, for the sake of this paper, it does not hurt to assume that $\mathcal{S}$ is just a (preordered) structural semiring, *i.e.*, that the monoidal structures are strict and that there is at most one arrow in every homset.

It is obvious how to interpret each type constructor of $\ell\mathcal{R}\mathsf{PCF}$ as a functor of $\mathcal{A}$. A typing derivation δ of the judgment $x_1 : B_1, \ldots, x_m : B_m, y_1 : [C_1]_{r_1}, \ldots, y_n : [C_n]_{r_n} \vdash e : A$ is interpreted by an arrow $[\![\delta]\!] : [\![B_1]\!] \otimes \cdots \otimes [\![B_m]\!] \otimes !_{r_1}[\![C_1]\!] \otimes \cdots \otimes !_{r_n}[\![C_n]\!] \to [\![A]\!]$ of $\mathcal{A}$, built by induction on the derivation:

– the interpretation of the rules id, lam and app is standard; the only non-standard feature is the $+$ operation on contexts, which is interpreted by means of the natural transformation c.
– The rule der corresponds to the natural transformation ε. The rule pr is just the application of the endofunctor $!_r(-)$, plus the natural transformations δ and m (if the context has more than one variable) or n (if the context is empty). The let rule is just a composition of morphisms.
– The sub rule is intepreted thanks to the contravariance of the action ! in its first argument: $p \preceq q$ corresponds to the existence of an arrow $f : p \to q$ in $\mathcal{S}$, from which we have an arrow $!_f(\mathrm{id}_A) : !_q A \to !_p A$ in $\mathcal{A}$, implementing subtyping. Free weakening is available because I is the terminal object of $\mathcal{A}$.

The intepretation of the type Nat and the PCF-specific constructions (successor, fixpoint. . .) require a suitable object and morphisms of $\mathcal{A}$, as usual. The interpretation of co-handlers is also dependent on the specific case, and cannot be defined in general.

Notice that, when $\mathcal{S}$ is the one-object category (which is tivially bimonoidal), then an exponential action is just a comonad $(!, \delta, \varepsilon)$, which is monoidal thanks to the natural transformations m and n. In this degenerated case, the conditions of Definition 13 boil down to asking that the natural transformations $\delta, \varepsilon, \mathsf{c}, \mathsf{w}$ are monoidal; that, for every object A of $\mathcal{A}$, $(!A, \mathsf{c}^A, \mathsf{w}^A)$ is a commutative comonoid in the category of free !-coalgebras of $\mathcal{A}$; and that free coalgebra morphisms (such as δ) are also comonoid morphisms. This amounts to giving a model of linear logic in the sense of [4] (and in fact, when $\mathcal{R}$ is the trivial semiring, $\ell\mathcal{R}\mathsf{PCF}$ is just multiplicative-exponential intuitionistic affine logic).

5 Examples

Before introducing the examples, it is worth noticing the importance of observations and coeffects for the evolution of observable quantities in the abstract machine states. First, note that the state changes only when an observation is performed. So, depending on where we place the observation we can obtain different quantitative information about our programs. Moreover, note that in the evaluation of a co-handler-free program the weight of each stack is always $\mathbf{1}$, so the state of the machine contains only an additive information of the shape $\mathbf{1} + \ldots + \mathbf{1}$, where the number of $\mathbf{1}$'s depends on the number of observations encountered in the evaluation. These two remarks are important for understanding the kind of analysis our framework can perform. Indeed, the observable quantities in the abstract machine states are ultimately the only quantities that the type system is able to analyze thanks to the soundness Theorem 10.

We show here the details of three examples and then we conclude by commenting on other examples. We choose three examples that stress different features of our framework: the first example is a classic of linear type systems—complexity analysis—this is helpful to see that we do not loose anything. The second example is inspired by Uustalu and Vene [37, 38] and Petricek *et al.* [32]—signal processing—this example requires an analysis that is quantitative on $\star$

but not on $+$, differing so from the previous one. Finally, the third example uses an operational semantics that is probabilistic—probability analysis—this shows that the analysis can be performed even when the underlying semantics changes.

In all cases, we will use one of the structural semirings introduced after Definition 1 and we will always use the same pole (or, rather, instances of a pole parametric in the semiring of choice). We first say that a state of the $\mathcal{K}^{\mathcal{R}}$ machine (C', p') is l-fixpoint-reachable from another state (C, p), and we write $(C, p) \twoheadrightarrow^{l} (C', p')$, if $(C, p) \to^{*} (C', p')$ and l is the number of f-transitions in the computation. Then, we set

$$\perp\!\!\!\perp_{(p,m)} := \{\ C \mid \text{whenever } (C, \mathbf{0}) \twoheadrightarrow^{l} (C', r),\ l < m \text{ implies } r \preceq p\ \}.$$

This pole expresses the following notion of correctness: a configuration is (p, m)-correct if, when evaluated with the quantity $\mathbf{0}$ and left evolving for a number of steps including strictly less than m recursive calls, produces an observable quantity bounded by p. We leave it as an easy (but instructive) exercise to the reader to show that the above definition gives a pole, for any structural semiring $\mathcal{R}$. The fact that it is adapted may be proved by a straightforward induction on types. Then, we have

Fact 15 *If $\vdash e : A$ with a type derivation δ, then any computation of the form $((e, [], \diamond), \mathbf{0}) \to^{*} (C, r)$ satisfies $r \preceq p[\delta]$.*

The above fact, which holds regardless of the chosen semiring, is an immediate consequence of Corollary 11 (we just spelled out the property $(e, [], \diamond) \in \perp\!\!\!\perp_{(p[\delta],\infty)}$ for this particular pole). The exact meaning will depend on the coeffects and on the semiring.

Complexity analysis. As a warm up we sketch how we can use observations to express a simple complexity analysis for coeffect-free programs inspired by [18, 9]. We want to analyze the complexity (in terms of time) of the execution of a closed term on the $\mathcal{K}^{\mathcal{R}}$ machine. We remark two properties of the machine: first, the evaluation of a program e in an empty environment and an empty stack requires environments containing only subterms of e; second, v-transitions are the only ones[5] that increase the overall size of a configuration[6]. Therefore, if e is a closed, observation- and co-handler-free term and n is the number of v-transitions in the computation from $(e, [], \diamond)$ to its normal form, a good estimate of the time complexity of such a computation is $n \cdot \mathsf{size}(e)$. This requires to compute n, which is precisely the quantity that our type system is able to provide.

First of all, we insert observations around each variable of e, obtaining a term e'. This does not alter the computational behavior of e but ensures that each o-transition is followed by an v-transition, so the number of o-transitions of e', which are the ones we can account for, bounds the number of v-transitions of e. Now, we set $\mathcal{R} := \overline{\mathbb{N}}$, with the usual operations and order. We obviously have

[5] Here we consider co-handler-free programs so there are no x-transitions.

[6] For a suitable notion of configurations size [9].

that $\vdash e : A$ implies $\vdash e' : A$. Call the latter type derivation δ. Recalling Fact 15, we have that any computation $((e', [], \diamond), \mathbf{0}) \to^* (C, n)$ satisfies $n \leq p[\delta]$. But, as noted above, n is nothing but the number of o-transitions performed in the computation, which in turn are no less than the v-transitions in the evaluation of e, so $p[\delta] \cdot \mathsf{size}(e)$ is the desired complexity bound.

We observe that our analysis for programs including recursion is very limited: the presence of fixpoints is likely to yield $p[\delta] = \infty$. However, as we will discuss below, we expect our approach to be adaptable to the use of dependent types as in [9].

Signal processing. The second example we consider is signal processing. We take this example from Petricek *et al.* [32] and show how $\ell\mathcal{R}\mathsf{PCF}$ provides a bound on the number of *look-ahead* operations each program performs for the given inputs. This enables optimization of memory allocation and buffering needs for each input. To make this example interesting, we add to the language a type Sig representing signals as globally clocked streams of natural numbers. We add to the grammar of $\ell\mathcal{R}\mathsf{PCF}$ terms non-denumerably many constants $s, s', \ldots$, one for each stream, and the nullary typing rule $\vdash s : \mathsf{Sig}$. We denote by $\underline{n} \cdot s$ the stream whose head is $\underline{n}$ and tail s.

We set $\mathcal{R} := \mathcal{A}$, the arctic semiring. We consider two co-handlers: read, with $A_{\mathsf{read}} = \mathsf{Sig}$, $B_{\mathsf{read}} = \mathsf{Nat}$ and $r_{\mathsf{read}} = 0$, and next, with $A_{\mathsf{next}} = \mathsf{Sig}$, $B_{\mathsf{next}} = \mathsf{Sig}$ and $r_{\mathsf{next}} = 1$. Their semantic maps are defined as follows: $\varphi_{\mathsf{read}}(\underline{n} \cdot s) = \underline{n}$ and $\varphi_{\mathsf{next}}(\underline{n} \cdot s) = s$. In other words, read and next return the head and tail of the stream, respectively. In order to check the semantic condition on φ_{read} and φ_{next} which ensure soundness, we need to define the realizability interpretation of the type Sig. This is done as for Nat: abusing the notations, we set $\mathsf{Sig} := \{\ ((s, []), (0, \infty)) \mid \forall \text{ streams } s\ \}$ and $\|\mathsf{Sig}\| := \mathsf{Sig}^{\perp\perp}$. Now, to prove the soundness of φ_{read}, we need to check that, for all $(\pi, (t, m)) \in \|\mathsf{Nat}\|^{\perp} = \mathsf{Nat}^{\perp}$, we have $(\mathsf{read}.\pi, (t, m)) \in \|\mathsf{Sig}\|^{\perp} = \mathsf{Sig}^{\perp}$. This amounts to checking that, for all $((s, []), (0, \infty)) \in \mathsf{Sig}$, we have $(s, [], \mathsf{read}.\pi) \in \perp\!\!\!\perp_{(w(\mathsf{read}.\pi)+t,m)}$. But $w(\mathsf{read}.\pi) = r_{\mathsf{read}} + w(\pi) = w(\pi)$ (remember that multiplication in $\mathcal{A}$ is addition in $\mathbb{N}$), so this follows by saturation. Similarly, for the soundness of next, let $(\pi, (t, m)) \in \mathsf{Sig}^{\perp}$. We need to check that $(\mathsf{next}.\pi, (t, m)) \in \mathsf{Sig}^{\perp}$, which amounts to verifying that, for every stack constant s, $(s, [], \mathsf{next}.\pi) \in \perp\!\!\!\perp_{(w(\mathsf{next}.\pi)+t,m)}$. Now, we know by saturation that $(s, [], \mathsf{next}.\pi) \in \perp\!\!\!\perp_{(w(\pi)+t,m)}$, but $w(\mathsf{next}.\pi) = 1 + w(\pi)$, so $\perp\!\!\!\perp_{(w(\pi)+t,m)} \subseteq \perp\!\!\!\perp_{(w(\mathsf{next}.\pi)+t,m)}$ by monotonicity, which allows us to conclude.

We are now in position to apply the Soundness Theorem 10. From $\vdash (\!|s|\!) : \mathsf{Sig}$ we have $((s, []), (t, \infty)) \in t \star \|\mathsf{Sig}\|$ for all $t \in \mathbb{N}$. Now, suppose that e is a program (with no observations) with a free variable x, and suppose that $x : [\mathsf{Sig}]_t \vdash e : A$ with a typing derivation δ. Theorem 10 gives us that $(e, [x/(\!|s|\!)], \diamond) \in \perp\!\!\!\perp_{(\max(p[\delta],t),\infty)}$. By observing the soundness function of Fig. 5, we realize that if e contains no observation, then $p[\delta]$ is necessarily $\mathbf{0}$, which is equal to $-\infty$ in $\mathcal{A}$. Therefore, $(e, [x/(\!|s|\!)], \diamond)$ is (t, ∞)-correct. So, any computation starting with $((e, [x/(\!|s|\!)], \diamond), -\infty)$ terminates on a state (C, u) such that $u \leq t$. By looking at the transition rules (Fig. 3), we see that u is the maximum (addition in $\mathcal{A}$ is max) of the number of next co-handlers that were present on the stack (multiplication in $\mathcal{A}$ is addition) at each o-transition. But since e contains no observation, o-transitions are possible only when we access the stack s in the environment, and it is not hard to see that the number of next co-handlers on the stack is the number of look-ahead operations performed on s. Therefore, we have

Fact 16 *If $x : [\mathsf{Sig}]_t \vdash e : A$, then e uses at most the first t values of the stack fed to its argument x.*

We hope that the above result gives an idea of the kind of analysis that may be performed for this application. Other more general results can be obtained by placing observations on different subterms. Also, here we considered only terminating computations but for this application one can use a different pole to allow analysis of non-terminating programs as well.

Probabilistic usage. We now turn to probabilistic setting. Monadic programming languages have been extensively used for describing probabilistic computations. Here we propose something slightly different. We want to consider the situation in which accessing certain memory locations is subjected to probabilistic failures. For this application, we set $\mathcal{R} = \overline{\mathbb{R}}^+$, the probability semiring, and we consider a single co-handler, coflip, with $A_{\mathsf{coflip}} = \mathbb{N}$, $B_{\mathsf{coflip}} = \mathbb{N}$ and $r_{\mathsf{coflip}} = \lambda \in [0, 1]$. The coeffect map $\varphi_{\mathsf{coflip}}$ is the identity, which is obviously sound. However, we change the operational semantics: the x-transition of the $\mathcal{K}^{\mathcal{R}}$ machine, when coflip is on the stack, is executed with probability λ, whereas with probability $1 - \lambda$ the machine halts because of a failure. When we want to model the fact that a variable x in a program e represents a failure-prone memory location, we replace every occurrence of x in e with $\mathsf{coflip}(x)$.[7]

Consider now a closed program e. We define $\mathsf{var}(e)$ as the number of v-transitions in the computation starting from $(e, [], \diamond)$ before a failure occurs or a normal form is reached. Due to the probabilistic nature of the machine, $\mathsf{var}(e)$ is a random variable with values in $\overline{\mathbb{N}}$. Our type system allows us to estimate the expected value of $\mathsf{var}(e)$. Indeed, given such a closed program e and a configuration C, we may define $\mathsf{obs}(e, C)$ as the number of o-transitions in the computation $(e, [], \diamond) \to^* C$, which is a random variable too. It is not hard to the check that the observable quantity $r \in \mathbb{R}^+$ of the computation $((e, [], \diamond), 0) \to^* (C, r)$ is the expected value of $\mathsf{obs}(e, C)$: every time an o-transition is executed, the quantity λ^n is added to the observable quantity, with n being the number of coflip coeffects on the stack. This is the probability of success, *i.e.*, the probability that the evaluation will "survive" in the current environment.

More generally, we define $\mathsf{obs}(e)$ as the number of o-transitions in the longest computation starting from $(e, [], \diamond)$, and we apply the same decoration used in complexity analysis, *i.e.*, we consider programs obtained by inserting observations around every variable of a program. In this way, we know that, if e' is the decoration of e, $\mathsf{var}(e) \leq \mathsf{obs}(e')$ and therefore, applying Fact 15, we have

[7] For simplicity we used a single co-handler, with a single probability of failure, but of course any number of co-handlers may be used, each with its own probability λ.

Fact 17 *Let e be a program (observation-free) and let $\vdash e : A$ through a typing derivation δ. Then, $p[\delta]$ bounds the expected value of* $\mathsf{var}(e)$.

Other analyses. We presented three examples that are representative of some of the reasoning that may be performed in our framework. Other analyses may be obtained in a similar way. For instance, a liveness analysis like the one of [32] may be obtained by considering the Boolean lattice. Furthermore, the scheduling analysis of [16], which uses a semiring of affine transformations, is another application of our framework, independently developed (we discuss this a bit more in Sect. 6).

More interestingly, also the type system for sensitivity analysis from [34] and the one for non-interference analysis of the SLam calculus [20] can be seen as instances of our calculus. Unfortunately, the realizability semantics is not enough expressive for proving the soundness of these analyses. What we need is a relational version of our realizability technique, which we leave for future investigations.

6 Related Work

Indexed notions of monads and comonads. Several works have extended monads with the aim of reasoning about more general effects: indexed monads [40], parametrised monads [2], layered monads [13], etc. Similarly, we aim at providing a theory to reason about general coeffects. Abadi *et al.* [1] use an indexed monad as a basis for their core calculus of dependencies. Similar to our modalities, their indexed monad is useful to capture dependencies between input and output and so to perform several program analysis. Moreover, they provide a generalized soundness result using domain theory. Tate [36] proposes the notion of *productor* to describe general producer effect systems. Interestingly, his notion can be specialized to capture all the other extensions of monads referred above. Tate also mention the notion of *consumptor*, as dual to productor, and he suggest as an example of consumptor the non-linear use of resources. Our development can be seen as a step in the direction of developing a theory of consumptor. Several effect type systems have been used for program analysis [35, 29]. The common aspect with our work is the use of indices in types to track information about the interaction of the program with the environment.

Uustalu and Vene [37, 38] have proposed a general comonadic approach to programming following the idea of values in context. In particular, they showed how to formulate several context dependent programming models in terms of comonadic computations. Extending this approach, the closest work to our own approach, in the motivations as well as in technical terms is certainly [32, 31]. In these papers, the authors present a coeffect system parametrised over a coeffect algebra that has some remarkable similarities with our notion of structural semiring. The main difference is that while we use two monoids for the different operations in the semiring—plus some additional structure—they use instead a semilattice and a monoid. The semilattice operation is idempotent, so they loose in this way the possibility of being quantitative with respect to this operation. Moreover, they consider "global" comonadic information, *i.e.*, applied to the whole context, whereas our information is "local", *i.e.*, on each variable, as customary in linear logic. The global aspect of their approach forces them to introduce an additional operation $\wedge$ in their algebra, needed to split the information in the case of λ-abstraction, which isolates one variable from the context. However, this operation has no algebraic requirement and in all of the examples they consider we are able to simulate it by means of subtyping. Finally, the most important difference is that they do not provide any soundness result for the analyses that may be performed using their framework (even though they do provide a categorical model), while we prove a parametric soundness theorem.

Linear indexed types. The idea of distinguishing between linear, single-use data and non-linear, reusable data has been one of the reason of the success of linear logic. Less attention has attracted the idea, already presented by Girard in [17], of using indexed approximated modalities $!_n$ for *counting* multiple uses of the same resource. The first real attempt on using this kind of modalities is Bounded Linear Logic [18], where modalities are indexed by polynomial expressions. More recently, indexed modalities similar to the ones studied in the present work have been used in [23, 9] for complexity analysis, in [34, 14] for sensitivity analysis in the context of Differential Privacy, and in [16] for automatic scheduling analysis.

Interestingly, the authors of the latter work, inspired by their previous work [15], introduce an abstract type system that is essentially the call-by-name fragment of our $\ell\mathcal{R}\mathsf{PCF}$. They also present a categorical model, less general than ours, and prove a coherence theorem for it. However, they do not prove any form of soundness. Therefore, the present paper and [16] (which will be presented at the same conference) are in a way complementary: that work provides a further, highly interesting example of analysis to which our soundness proof applies.

Other indexed modalities, dubbed "subexponentials", have also been used in [30] with the aim of increasing the expressiveness of linear logic programming. However, this use of indexes seems orthogonal to the one studied in the present paper. Another work that, superficially, seems to be related to ours is [24], where the authors introduced a class of denotational models of linear logic parametrized over continuous semirings. However, the connection does not seem to be very strong, because their technical results are totally different. Indeed, the elements of their semiring are used as coefficients for terms in the language.

Realizability and logical relations. Realizability and (unary) logical relations are well-establishe reasoning tools. Step-indexing and biorthogonality (also referred as "TT closure") are technical mechanisms that permit to extend the reasoning to consider languages with potentially infinite computations and to consider different properties of programs in a uniform way. Several works have studied step indexing and biorthogonality, see for instance [5, 33]. The extension of realizability for reasoning about quantitative properties has been pioneered in [21] and further developed in [10, 11]. The realizability we use is in the spirit of the quantitative classical realizability proposed by Brunel [7]. Moreover, we use a combination of quantitative realizability with biorthogonality and step indexing similar to the one of [8] with the difference that here the step-indexing is used as usual to control recursion. A last work that has some similarities with ours is [3]. The authors use indexed types, logical relations and parametricity to achieve invariance under changes of data representation. The invariance allows them to capture program properties in a spirit similar to ours. In particular, also their language is parametrized over a choice of basic data types and primitives.

7 Conclusion and Future Work

In our work we focused on showing how indexed linear types, naturally arising from linear logic semantics, can be used to talk about value in context. This suggests that linear logic semantics can be a unifying framework for several program analyses. Several steps however need to be done. A first simple step is to lift the analysis we have presented here to the standard lambda calculus—this can be done in a rather standard way by using the usual call-by-name and call-by-value translation. Moreover, a type inference algorithm parametrized over constraints in $\mathcal{R}$ can be designed in a natural way following D'Antoni *et al.* [12]. A second and more important step is to broaden the scope of the analysis. Indeed, the analysis for the fixpoint and pattern matching are too limited in practice. For this reason we plan to extend our work with polymorphism and a restricted form of dependent types as in [9, 14]. A point that is worth to stress here is that the realizability semantics we have presented is already able to accommodate some of the future presented there. For instance by interpreting basic constants with non-zero quantities we can accommodate basic indexed types.

Acknowledgments. We are grateful to Emilio Jesús Gallego Arias, Jan Hoffmann, Paul-André Melliès, Dominic Orchard, and Tarmo Uustalu for fruitful discussions. This work benefited from partial support of: ANR, under projects Logoi ANR-2010-BLAN-0213-02 (A. Brunel and D. Mazza) and Coquas ANR-12-JS02-006-01 (D. Mazza); the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement No. 272487 (M. Gaboardi); DARPA Crash program under Contract No. FA8650-10-C-7090 (A. Brunel). The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL*. ACM, 1999.
[2] R. Atkey. Parameterised notions of computation. *JFP*, 19(3-4), 2009.
[3] R. Atkey, P. Johann, and A. Kennedy. Abstraction and invariance for algebraically indexed types. In *POPL*. ACM, 2013.
[4] N. Benton, G. M. Bierman, J. M. E. Hyland, and V. de Paiva. Term assignment for intuitionistic linear logic. Technical Report 262, University of Cambridge, 1992.
[5] N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In *TLDI*. ACM, 2009.

[6] N. Benton and P. Wadler. Linear logic, monads and the lambda calculus. In *LICS*. IEEE, 1996.
[7] A. Brunel. Quantitative classical realizability. *Inf. and Comp.*, 2013. To appear.
[8] A. Brunel and A. Madet. Indexed realizability for bounded-time programming with references and type fixpoints. In *APLAS*. Springer, 2012.
[9] U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In LICS. IEEE, 2011.
[10] U. Dal Lago and M. Hofmann. Bounded linear logic, revisited. In *TLCA*. 2009.
[11] U. Dal Lago and M. Hofmann. Realizability models and implicit complexity. *TCS*, 412(20), 2011.
[12] L. D'Antoni, M. Gaboardi, E. J. Gallego Arias, A. Haeberlen, and B. Pierce. Sensitivity analysis using type-based constraints. In *FPCDSL*. ACM, 2013.
[13] A. Filinski. Representing layered monads. In *POPL*. ACM, 1999.
[14] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *POPL*. ACM, 2013.
[15] D. R. Ghica and A. Smith. Geometry of synthesis III: Resource management through type inference. In *POPL*. ACM, 2011.
[16] D. R. Ghica and A. Smith. Bounded linear types in a resource semiring. In *ESOP*. Springer, 2014.
[17] J.-Y. Girard. Linear logic. *TCS*, 50(1):1–102, 1987.
[18] J.-Y. Girard, A. Scedrov, and P. Scott. Bounded linear logic. *TCS*, 97(1), 1992.
[19] B. Guillou. Strictification of categories weakly enriched in symmetric monoidal categories. *Theory and Applications of Categories*, 24(20):564–579, 2010.
[20] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *POPL*. ACM, 1998.
[21] M. Hofmann and P. J. Scott. Realizability models for BLL-like languages. *TCS.*, 318(1-2), 2004.
[22] J.-L. Krivine. A call-by-name lambda-calculus machine. *HOSC*, 20(3), 2007.
[23] U. D. Lago and U. Schöpp. Functional programming in sublinear space. In *ESOP*. Springer, 2010.
[24] J. Laird, G. Manzonetto, G. McCusker, and M. Pagani. Weighted relational models of typed lambda-calculi. In *LICS*. IEEE, 2013.
[25] M. Laplaza. Coherence for distributivity. *Lecture Notes in Math.*, 281, 1972.
[26] P.-A. Melliès. Categorical semantics of linear logic. *Panoramas et Syntheses*, 2009.
[27] P.-A. Melliès. Parametric monads and enriched adjunctions. Technical report, 2012. http://www.pps.univ-paris-diderot.fr/ mellies/tensorial-logic/.
[28] E. Moggi. Computational lambda-calculus and monads. In *LICS*. IEEE, 1989.
[29] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
[30] V. Nigam and D. Miller. Algorithmic specifications in linear logic with subexponentials. In *PPDP*. ACM, 2009.
[31] D. Orchard. Programming contextual computations, 2013. Cambridge University.
[32] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: Unified static analysis of context-dependence. In *ICALP*, 2013.
[33] A. M. Pitts. Step-indexed biorthogonality: a tutorial example. Dagstuhl, 2010.
[34] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ICFP*. ACM, 2010.
[35] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *LICS*. IEEE, 1992.
[36] R. Tate. The sequential semantics of producer effect systems. In *POPL*, 2013.
[37] T. Uustalu and V. Vene. Signals and comonads. *J. UCS*, 11(7), 2005.
[38] T. Uustalu and V. Vene. Comonadic notions of computation. *ENTCS*, 203, 2008.
[39] P. Wadler. The essence of functional programming. In *POPL*. ACM, 1992.
[40] P. Wadler. The marriage of effects and monads. In *ICFP*. ACM, 1998.