

Linear Effects, Exceptions, and Resource Safety

A Curry-Howard Correspondence for Destructors

Sidney Congard^{1,2}, Guillaume Munch-Maccagnoni¹, and Rémi Douence²

¹ INRIA, LS2N CNRS, Nantes, France

² IMT Atlantique, Nantes, France

October 27, 2025

Abstract. We analyse the problem of combining linearity, effects, and exceptions, in abstract models of programming languages, as the issue of providing some kind of strength for a monad $T(- \oplus E)$ in a linear setting. We consider in particular for T the *allocation monad*, which we introduce to model and study resource-safety properties. We apply these results to a series of two linear effectful calculi for which we establish their resource-safety properties.

The first calculus is a linear call-by-push-value language with two allocation effects **new** and **delete**. The resource-safety properties follow from the linear (and even ordered) character of the typing rules.

We then explain how to integrate exceptions on top of linearity and effects by adjoining default destruction actions to types, as inspired by C++/Rust destructors. We see destructors as objects $\delta : A \rightarrow TI$ in the slice category over TI . This construction gives rise to a second calculus, an *affine* ordered call-by-push-value language with exceptions and destructors, in which the weakening rule performs a side-effect. As in C++/Rust, a “move” operation is necessary to allow random-order release of resources, as opposed to last-in-first-out order. Moving resources is modelled as an exchange rule that performs a side-effect.

1 Introduction

The application of monads to study effects in programming languages [36, 37, 14, 46, 43, 29, 28], as well as the application of linearity to study resource-sensitive aspects of computation [17, 27, 3, 13, 31, 5, 24], are well-established. However, the combination of effects and resources, despite receiving some attention [47, 20, 21, 48, 34, 35, 10], has much less developed theory and case studies.

In order to understand why the combination of effects and resources poses new challenges, it is useful to remind that a monad T modelling computational effects is given by an endofunctor on a cartesian category $\mathcal{C}$, together with families of maps

$$\eta_A : A \rightarrow TA \quad \text{and} \quad \mu_A : TTA \rightarrow TA \quad (1)$$

natural in $A \in \mathcal{C}$ and satisfying monoid-like laws, and together with a family of maps called strength

$$\sigma_{\Gamma, A} : \Gamma \times TA \rightarrow T(\Gamma \times A) \quad (2)$$

natural in $\Gamma \in \mathcal{C}$ and $A \in \mathcal{C}$ and satisfying four laws stating the compatibility with η, μ and with the monoidal structure induced by $\times$ and 1 . The general principle is that two typed expressions of type $\Gamma \vdash A$ and $\Gamma, A \vdash B$ are allowed to compose as follows:

$$\Gamma \vdash t : A \quad \text{and} \quad \Gamma, x : A \vdash u : B \quad \Rightarrow \quad \Gamma \vdash \text{let } x = t \text{ in } u : B \quad (3)$$

which is reflected in their interpretation as morphisms in the Kleisli category of T

$$\llbracket t \rrbracket : \Gamma \rightarrow TA \quad \text{and} \quad \llbracket u \rrbracket : \Gamma \times A \rightarrow TB$$

through the ability to select x in u where the strength plays an essential role:

$$\llbracket \text{let } x = [] \text{ in } u \rrbracket : \Gamma \times TA \xrightarrow{\sigma_{\Gamma, A}} T(\Gamma \times A) \xrightarrow{Tu} TTB \xrightarrow{\mu_B} TB. \quad (4)$$

It is also useful to remind that the application of linearity to model resource-sensitive phenomena of data and computation requires to move from a cartesian category $(\mathcal{C}, \times, 1)$ to a symmetric monoidal category $(\mathcal{C}, \otimes, I)$. This modification gives control over duplication and erasure (corresponding to contraction and weakening in logic) since the maps

$$A \rightarrow A \otimes A \quad \text{and} \quad A \rightarrow I \quad (5)$$

are no longer available for a general A . The linear logic viewpoint nevertheless subsumes the cartesian viewpoint using a resource modality “!”, which is a comonad such that its Eilenberg-Moore category $\mathcal{C}^!$ has a symmetric monoidal structure that coincides with the symmetric monoidal structure of $\mathcal{C}$ in terms of the underlying objects, and such that this symmetric monoidal structure on $\mathcal{C}^!$ is cartesian [32, 33]. Concretely, duplication and erasure (5) are available whenever A is given with a map $A \rightarrow !A$ satisfying two laws of compatibility with the comonad structure. More rarely—but importantly in our story—the symmetric monoidal structure is sometimes replaced by a monoidal structure, which amounts to providing control over the exchange rule in logic since the symmetry

$$A \otimes B \rightarrow B \otimes A \quad (6)$$

is no longer available unconditionally. The corresponding logics are called *ordered* instead of linear. But here again, the exchange rule can be reintroduced selectively with a resource modality [19]. Ordered logic can be useful to model the order in the release of resources [44, 45, 55], as we will see.

1.1 Linear effects

In this context, it is tempting to define a notion of *linear computational effect* to be given again by a monad (T, η, μ) (1) on $\mathcal{C}$ and a strength now given as a family of maps

$$\sigma_{\Gamma, A} : \Gamma \otimes TA \rightarrow T(\Gamma \otimes A) \quad (7)$$

natural in Γ and A and satisfying the same four laws. It is also called a *left strength* when the monoidal structure $\otimes$ is non-symmetric. An example of such a strong monad

is the linear state monad $S \multimap (- \otimes S)$ for any $S \in \mathcal{C}$ in any symmetric monoidal closed category $\mathcal{C}$. This transposition to the linear context of computational effects is well-behaved and works similarly to (non-linear) monadic effects [46, 20].

Unfortunately, important examples of effects in models of linear logic do not have a strength (7) in such a restrictive sense, such as the control effects modelled by the following monads in models of linear logic [7, 20, 21]:

- the monad “?” of *linearly-defined continuations* (but not *linearly-used*) $?A \stackrel{\text{def}}{=} !(A \multimap \perp) \multimap \perp$, whose effect corresponds to call/cc-style control operators.
- for any object E , the exception monad $\mathcal{E} \stackrel{\text{def}}{=} - \oplus E$, which is used to model error types and exceptions.

At this point, it is instructive to go back to the computational intuition behind the strength with the interpretation of the monadic binding (4, where $\times$ is now replaced by $\otimes$). We can see that the parameter Γ in (7) represents the context of variables in the rest of the computation (let $x = []$ in u) at the location where an effect is performed. But control effects have this particularity that they can change how many times the rest is computed, so we cannot guarantee that variables in Γ are not duplicated nor erased.

1.2 Strength with respect to a resource modality

In fact, starting from this intuition we can suggest a relaxed notion of strength, something like (7) where Γ is assumed to possess a coalgebra structure $\tau : \Gamma \rightarrow !\Gamma$, so that it can be erased. It so happens ([7, 20, 21]) that the monad ? of linearly-defined continuations and the exception monad $\mathcal{E}$ each do have a (*left*) *strength with respect to $U^!$* , where $U^! : \mathcal{C}^! \rightarrow \mathcal{C}$ is the forgetful functor of the category of algebras,³ defined as a family of maps

$$\sigma_{\Gamma, A} : U^! \Gamma \otimes T A \rightarrow T(U^! \Gamma \otimes A) \quad (8)$$

natural in $\Gamma \in \mathcal{C}^!$ and $A \in \mathcal{C}$ and satisfying again four laws of compatibility with η , μ and with the monoidal structures of $\mathcal{C}$ and $\mathcal{C}^!$. If we look concretely at the strength with respect to $U^!$ of the exception monad $\mathcal{E}$, we see that it consists for each $A \in \mathcal{C}$ and $\Gamma \in \mathcal{C}^!$ in two maps

$$U^! \Gamma \otimes A \rightarrow (U^! \Gamma \otimes A) \oplus E \quad \text{and} \quad U^! \Gamma \otimes E \rightarrow (U^! \Gamma \otimes A) \oplus E \quad (9)$$

corresponding respectively to the *normal case* and the *exceptional case* describing the propagation of the exception, whose copairing is subject to the mentioned naturality and coherence conditions. The normal case is given by the left inclusion whereas the exceptional case is obtained from $U^! \Gamma \otimes E \rightarrow E$, which follows from the erasure (5) under the hypothesis that the monoidal structure on $\mathcal{C}^!$ is cartesian. We are now ready to make an important observation: for the purpose of deriving a strength (8) we could consider other resource modalities than !; for instance the hypothesis that $\mathcal{C}^!$ is semi-cartesian suffices (the monoidal unit of $\mathcal{C}^!$ being a terminal object). This amounts to

³ The cited authors actually consider this notion for the Kleisli adjoint resolution of !, but it can be defined similarly for any adjoint resolution, and even any strong monoidal functor U .

replacing $!$ in (8) with an affine resource modality that permits erasure (weakening) but not duplication in (5).

Now that we have described a relaxed notion of strength for the monad of linearly-defined continuations and the exception monad in a linear context, we have to mention how restrictive a strength (8) seems to be in light of the interpretation of composition (3). The notion of strength with respect to $!$ amounts to precluding any linear variable from appearing in the monadic binding (4), in other words linear variables can only appear if it can be ensured that no control effect is performed. This models a restrictive approach in which linearity and control effects are exclusive with one another.

1.3 Destructors

It is now clear that linearity and control effects do not mix well, or so it seems. Our goal is to show how linearity and exceptions can actually be mixed in more important situations than it seems, with a technique that we explain abstractly but that has been discovered from practical consideration in the context of resource management by the conceptors of the C++ programming language [26, 49, 22]. (We are deferring the more historical discussion to section 6.)

The starting point for describing this important discovery is to assume a linear model of computation as above given by a symmetric monoidal closed category $\mathcal{C}$ together with a given notion of linear effect, that is, a given monad T on $\mathcal{C}$ with a strength (7). As we have explained, this is a notion of linear model with linear effect which has previously been studied [46, 20, 34]; in particular a concrete model of computation is given by a *linear call-by-push-value* calculus [10] that will provide the basis of a first linear calculus we study.

In this model, we are interested in integrating exceptions. Assuming $\mathcal{C}$ has finite coproducts and that tensor products distribute over the coproducts, this means that we want to combine the monad T and the exception monad $\mathcal{E}$ into a monad $T\mathcal{E}$. Remember that as a general principle of the exception monad, $T\mathcal{E}$ has a monad structure arising from a distributive law of monads $\mathcal{E}T \rightarrow T\mathcal{E}$. However we meet again the obstacle that $\mathcal{E}$ does not have a strength in general, so we cannot obtain a strength for $T\mathcal{E}$ by composition.

But it might still be the case that $T\mathcal{E}$ has some kind of strength. This amounts to finding for all $A \in \mathcal{C}$ something like

$$\Gamma \otimes T(A \oplus E) \rightarrow T((\Gamma \otimes A) \oplus E) \quad (10)$$

for which, given that T is strong, it suffices to find maps

$$\Gamma \otimes A \rightarrow T((\Gamma \otimes A) \oplus E) \quad \text{and} \quad \Gamma \otimes E \rightarrow T((\Gamma \otimes A) \oplus E)$$

subject to some conditions. The one on the left-hand side—the normal return—is obtained in a straightforward manner by inclusion and unit of T . For the one on the right-hand side—the exceptional return—observe that the difference with (9) is the presence of T on the right-hand side. The second signature suggests that Γ has to be erasable, but that erasure is allowed to perform effects in T . Indeed, observe that the second map above can be derived if Γ is provided with some map $\Gamma \rightarrow TI$. We call such an erasure map that performs an effect a *destructor*, by analogy with C++ destructors.

1.4 The monoidal category of destructors

The desire to find a strength for $T\mathcal{E}$ therefore suggests to consider for Γ in (10) an object with a given destructor $\delta : \Gamma \rightarrow TI$, that is to say an object in the slice category $\mathcal{C}_{/TI}$. Recall that it is the category whose objects are arbitrary pairs $(A \in \mathcal{C}, \delta : A \rightarrow TI)$, and whose morphisms are morphisms in $\mathcal{C}$ that preserve the second component δ . The slice category $\mathcal{C}_{/TI}$ enjoys a series of nice properties, the most striking one being that it gives rise to a resource modality on $\mathcal{L}$ [9].

- $\mathcal{C}_{/TI}$ has a monoidal structure arising from the monoid structure on TI :

$$TI \otimes TI \xrightarrow{\sigma_{TI,I}} T(TI \otimes I) \xrightarrow{\cong} TTI \xrightarrow{\mu} TI$$

The monoidal unit is given by $(I, \eta_I : I \rightarrow TI)$ and, for $(A, \delta_A : A \rightarrow TI)$ and $(B, \delta_B : B \rightarrow TI)$, the tensor product is given by:

$$(A \otimes B, \delta_{A \otimes B} : A \otimes B \xrightarrow{\delta_A \otimes \delta_B} TI \otimes TI \rightarrow TI).$$

In words, $A \otimes B$ has a canonical destructor which releases B and A , as we will see in the reverse order of allocation.

- We also observe that there is a strong monoidal functor $U : \mathcal{C}_{/TI} \rightarrow \mathcal{C}$, that sends the monoidal structure of $\mathcal{C}_{/TI}$ to the one of $\mathcal{C}$ (strictly so). It is a nice exercise to check that the monad $T\mathcal{E}$ is indeed strong with respect to U .
- If we assume that $\mathcal{C}$ has finite products that we note $(\&, \top)$, then U has a right adjoint $G : \mathcal{C} \rightarrow \mathcal{C}_{/TI}$ sending objects $A \in \mathcal{C}$ to

$$(A \& TI, \pi_2 : A \& TI \rightarrow TI) \in \mathcal{C}_{/TI} \quad (11)$$

and morphisms $f : A \rightarrow B$ to $f \& TI$. In particular if we note $\mathcal{D} \stackrel{\text{def}}{=} UG$ the comonad on $\mathcal{C}$ associated to the adjunction, one has $\mathcal{D} = (- \& TI)$.

- The latter adjunction is comonadic, that is, $\mathcal{C}_{/TI} \simeq \mathcal{C}^{\mathcal{D}}$. This can be observed from the fact that a coalgebra $(A, \tau : A \rightarrow \mathcal{D}A)$ for $\mathcal{D}$ is essentially the same thing as an object A provided with a map $A \rightarrow TI$ with no other condition.

At this stage we find it useful to sum up the assumptions on $\mathcal{C}$: a distributive symmetric monoidal category with finite products and a chosen strong monad T . In particular any model of linear logic or intuitionistic linear logic is suitable (where T can be any linear state monad, for instance). We have just established that in those models, $\mathcal{D}$ has the structure of a resource modality whose Eilenberg-Moore category is (equivalent to) $\mathcal{C}_{/TI}$, and for which the monad $T\mathcal{E}$ is strong with respect to $U : \mathcal{C}_{/TI} \rightarrow \mathcal{C}$. However, unlike usual resource modalities that add duplication and/or erasure (5), $\mathcal{C}_{/TI}$ is in general not cartesian nor even semi-cartesian. In fact its monoidal structure is not even symmetric (unless T is commutative). Indeed, $\delta_{A \otimes B}$ and $\delta_{B \otimes A}$ can in general be distinguished by the order in which the effects of δ_A and δ_B are performed.

Now, just as linear models with a resource modality “!” give rise to an intuitionistic model by the Girard translations [17], one can build an *ordered call-by-push-value* model whose (positive) objects are those of $\mathcal{C}_{/TI}$, as an instance of a general principle generalising the Girard translation [34, 10] applied to the resource modality $\mathcal{D}$. This motivates the study of a second calculus after the first one, whose (positive) types are supplied with a destructor and whose effects include those of T and exceptions.

1.5 Modelling resources with the allocation monad

In this new context, notice that everything is linear in the sense that we did not make use of a resource modality “!”. We set out to show how *resource-safety* properties are thus ensured by construction. *Resource management* in programming aims to ensure the correct allocation, use, and release of *resources*, which are transient values denoting the validity of some state (a memory allocation, a lock, or typically resources from the operating system). The need for correct resource management in programming compounds the problems with mixing linearity and control: indeed, a need to handle errors arises from the possibility that resource acquisition can fail, whereas a need for linearity arises from the fact that the resources must be released in a timely fashion and no more than once.

We introduce the *allocation monad* as a way to model resource management and state and establish resource-safety properties. The allocation monad will play the role of T in the slice category $\mathcal{C}_{/TI}$. We first assume given an atomic type R of resources and ask for two effectful operations:

- **new** : $I \multimap R \oplus I$ acquires a resource, or fails if there are no resources available. Later on, it will also be given with type $I \multimap R$ when the failure can be represented with an exception.
- **delete** : $R \multimap I$ releases the given resource.

Notice that no operation is given for interacting with resources: we are only interested in observing resource-safety properties of the program state arising from the (correct) use of **new** and **delete**. Notice also that we represent failure with an explicit use of the exception monad $\mathcal{E} = - \oplus E$ for $E \stackrel{\text{def}}{=} I$ as an error type. But using an error type explicitly is not a solution to the lack of exceptions since we can expect similar obstacles to programming to arise from the lack of general strength for $\mathcal{E}$ (see discussion in section 6).

To implement those effectful operations, we then define the allocation monad as the linear state monad on the type $[R]$ of lists of R :

$$TA \stackrel{\text{def}}{=} [R] \multimap A \otimes [R]$$

As an instance of the linear state monad, it is strong.

- **new** : $I \rightarrow T(R \oplus I)$ is defined by popping an element from the list, or returning the error I if the list is empty.
- **delete** : $R \rightarrow TI$ is defined by pushing the resource onto the list; this is guaranteed to never fail.

Observe that a final program is of type $I \rightarrow ([R] \multimap A \otimes [R])$. Its execution consists in supplying an initial list of available resources (*freelist*). Now if the type A is purely positive and does not contain the type R , we expect that *a correct linear program will have as final freelist the same list as given initially, up to a permutation*. We can understand this as a resource-safety property: only resources acquired from the initial freelist have been released, all resources have eventually been released including in case of error; no resource has been released twice.

Moreover, assuming now an ordered rather than linear setting, that is without symmetry (6), we expect that *the final freelist is identical to the initial freelist*. This reflects the

observation that without symmetry, resources are managed in a last-in-first-out (LIFO) order. Ordered logics often have two forms of closures operating on opposite sides $B \multimap A$ and $A \multimap B$, but as we will see this ordered resource-safety property is only true when one does not have $A \multimap B$; hence we only have $B \multimap A$ in this paper for the ordered logics (which still coincides with $A \multimap B$ when adding symmetry).

1.6 Outline and contributions

We have just explained abstractly how to mix linearity and exceptions, and introduced the *allocation monad* modelling an idealised global “freelist” allocator for studying resource-safety properties.

In **Section 2** we define a *linear call-by-push-value* calculus $\mathcal{L}$ which has an *ordered* fragment by removing its exchange rule (6), hence named $\mathcal{O}$. This calculus features allocation effects (**new**, **delete**), together with a simple type system and a small-step operational semantics. We establish elementary properties of $\mathcal{L}$ and $\mathcal{O}$ relating typing to reduction.

In **Section 3** we define and prove *resource-safety properties* based on the allocation monad: a (well-typed) ordered program releases all its resources exactly once in reverse order of allocation; a (well-typed) linear program does so in random order.

In **Section 4** we present an extension of the first language: a calculus $\mathcal{O}_{\mathcal{E},\text{move}}$ which is an ordered call-by-push-value language with allocation effects and exceptions, in which every positive type has a chosen effectful erasure map, and in which allocation raises an exception upon failure. It can explicitly exchange variables outside of its ordered fragment $\mathcal{O}_{\mathcal{E}}$. We establish that this language also ensures the resource-safety properties for (well-typed) programs, by describing a translation into the calculus $\mathcal{L}$ that preserves the ordered fragment. Schematically:

$$\begin{array}{ccc} \mathcal{O}_{\mathcal{E}} & \xrightarrow{\llbracket - \rrbracket} & \mathcal{O} \\ \cap & & \cap \\ \mathcal{O}_{\mathcal{E},\text{move}} & \xrightarrow{\llbracket - \rrbracket} & \mathcal{L} \end{array}$$

Then, in **Section 5** we define a presheaf model for $\mathcal{O}$, in which central types coincide with resource-free types and with types with a trivial destructor. It models an ordered logic with a destructor for all types without being a model of affine logic in the usual sense.

Lastly, in **Section 6**, we conclude with a discussion replacing the results in the context of the theory of programming languages.

2 An ordered effectful calculus of allocations

After Moggi’s use of strong monads to model computational effects [37], investigations on the decomposition of the monad as an adjunction led to call-by-push-value (Levy [28]), which decomposes call-by-name and call-by-value semantics with two kinds that classify types as positive or negative. Linear call-by-push-value [10] refines call-by-push-value

by moving from a cartesian setting to a linear setting, and by generalising the Girard translations of intuitionistic logic into linear logic.

To model effects described by the allocation monad, we devise a calculus $\mathcal{O}$, which refines linear call-by-push-value into an *ordered* call-by-push-value. Operations **new** and **delete** are the two available effects. The calculus is given typing rules corresponding to an ordered (or non-commutative) intuitionistic logic, and a small-step operational semantics on closed terms with an abstract machine that makes explicit the state of the freelist. In this calculus where the order of variables in the context matters, functions bind the leftmost variable and let bindings bind the rightmost variable.

The calculus $\mathcal{L}$ consists in $\mathcal{O}$ plus the exchange rule.

2.1 Grammar

Two syntactic categories define negative and positive types, following the connective polarities:

$$\text{Types } A, B : \begin{cases} \text{Positive types} & P, Q ::= R \mid 1 \mid A \otimes B \mid A \oplus B \\ \text{Negative types} & N, M ::= B \multimap A \mid A \& B \end{cases}$$

We introduce the notation $\varepsilon ::= + \mid -$ for polarities, and associate to each type its polarity $\varpi(A)$ with $\varpi(P) = +$, $\varpi(N) = -$.

Our untyped expressions do not have typing annotations. Instead, expressions that match elimination rules and let bindings get polarity annotations to retain information about their evaluation strategy. Values are substitutable expressions: they are made of variables, all negative expressions (as they follow the call-by-name evaluation strategy) and positive expressions that have the shape of values in call-by-value. Pattern matchings for positive values are written as δ expressions: unit, strict products and sums, while lazy pair projections and function applications retain the usual notation from lambda-calculus.

$$\begin{array}{ll} \text{Expressions} & t, u ::= v \mid (\text{let } x^+ = t \text{ in } u)^+ \mid (\text{let } x^- = v \text{ in } u)^+ \mid \delta(v, (x, y).t)^+ \mid \\ & \delta(v, ().t)^+ \mid \delta(v, x.t, y.u)^+ \mid (vw)^+ \mid (\pi_1 v)^+ \mid (\pi_2 v)^+ \\ \text{Values} & v, w ::= (\text{let } x^+ = t \text{ in } v)^- \mid (\text{let } x^- = v \text{ in } w)^- \mid \delta(v, (x, y).w)^- \mid \\ & \delta(v, ().w)^- \mid \delta(v, x.w, y.w')^- \mid (vw)^- \mid (\pi_1 v)^- \mid (\pi_2 v)^- \mid \\ & x \mid \mathbf{new} \mid \mathbf{delete} \mid (v, w) \mid () \mid \iota_1 v \mid \iota_2 v \mid \lambda x.t \mid \langle t, u \rangle \mid r_{n \in \mathbb{N}} \end{array}$$

2.2 Typing rules

We define in fig. 1 typing rules with restrictions to values. This restriction is only meaningful for positive types, as all negative expressions are values. This simplifies semantics and prevents from encoding arbitrary choices about the evaluation order of expressions in typing rules, *e.g.* whether starting by evaluating the first or second component of a pair. Type polarity annotations A_ε in typing rules assert that $\varpi(A) = \varepsilon$. Contexts Γ, Δ are lists of typed variables. $\Sigma(\Gamma, \Gamma')$ is the set of maps from Γ to Γ' made of permutations and renamings. $t[\sigma]$ is a meta-notation that denotes t where all variables have been renamed following σ .

$$\begin{array}{c}
\frac{}{x : A \vdash x : A} \text{var} \qquad \frac{\Gamma \vdash t : A \quad \sigma \in \Sigma(\Gamma, \Gamma')}{\Gamma' \vdash t[\sigma] : A} \text{struct} \\
\frac{}{\vdash \mathbf{new} : (R \oplus 1) \multimap 1} \text{new} \qquad \frac{}{\vdash \mathbf{delete} : 1 \multimap R} \text{delete} \\
\frac{\Delta \vdash t : A_\epsilon \quad \Gamma, x : A \vdash u : B_{\epsilon'}}{\Gamma, \Delta \vdash (\text{let } x^\epsilon = t \text{ in } u)^{\epsilon'} : B} \text{let} \\
\frac{\Gamma \vdash v : A \quad \Delta \vdash w : B}{\Gamma, \Delta \vdash (v, w) : A \otimes B} \otimes_i \quad \frac{\Delta \vdash v : A \otimes B \quad \Gamma, x : A, y : B, \Gamma' \vdash t : C_\epsilon}{\Gamma, \Delta, \Gamma' \vdash \delta(v, (x, y), t)^\epsilon : C} \otimes_e \\
\frac{}{\vdash () : 1} 1_i \qquad \frac{\Delta \vdash v : 1 \quad \Gamma, \Gamma' \vdash t : A_\epsilon}{\Gamma, \Delta, \Gamma' \vdash \delta(v, (), t)^\epsilon : A} 1_e \\
\frac{\Gamma \vdash v : A}{\Gamma \vdash \iota_1 v : A \oplus B} \oplus_{i1} \quad \frac{\Gamma \vdash v : B}{\Gamma \vdash \iota_2 v : A \oplus B} \oplus_{i2} \\
\frac{\Delta \vdash v : A \oplus B \quad \Gamma, x : A, \Gamma' \vdash t : C_\epsilon \quad \Gamma, y : B, \Gamma' \vdash u : C}{\Gamma, \Delta, \Gamma' \vdash \delta(v, x, t, y, u)^\epsilon : C} \oplus_e \\
\frac{x : A, \Gamma \vdash t : B}{\Gamma \vdash \lambda x. t : B \multimap A} \multimap_i \quad \frac{\Gamma \vdash w : A \quad \Delta \vdash v : B_\epsilon \multimap A}{\Gamma, \Delta \vdash (vw)^\epsilon : B} \multimap_e \\
\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \& B} \&_i \quad \frac{\Gamma \vdash v : A_\epsilon \& B}{\Gamma \vdash (\pi_1 v)^\epsilon : A} \&_{e1} \quad \frac{\Gamma \vdash v : A \& B_\epsilon}{\Gamma \vdash (\pi_2 v)^\epsilon : B} \&_{e2}
\end{array}$$

Fig. 1. Typing rules for $\mathcal{L}$

Polarity annotations of expressions are inferred from the type polarities in typing rules. Hence, we will leave them implicit for typed expressions. The following notations allow to define and type expressions without restrictions with respect to values, by picking an arbitrary order to evaluate expressions. Such expressions may not always be well-typed in $\mathcal{O}$, since let bindings can only bind the right-most variable.

$$\begin{array}{ll}
\iota_i t^* \stackrel{\text{def}}{=} \text{let } x = t \text{ in } \iota_i x & \delta^*(t, (x, y), u) \stackrel{\text{def}}{=} \text{let } z = t \text{ in } \delta(z, (x, y), u) \\
(t, u)^* \stackrel{\text{def}}{=} \text{let } x = t \text{ in let } y = u \text{ in } (x, y) & \delta^*(t, x, u, y, u') \stackrel{\text{def}}{=} \text{let } z = t \text{ in } \delta(z, x, u, y, u') \\
(vt)^* \stackrel{\text{def}}{=} \text{let } x = t \text{ in } vx & t; u \stackrel{\text{def}}{=} \text{let } x = t \text{ in } \delta(x, (), u)
\end{array}$$

As an example, we can define in $\mathcal{O}$ the following program that allocates then frees two resources. On allocation failure of s , it is forced to free r before returning:

$$\vdash \delta^*(\mathbf{new}(), r, \delta^*(\mathbf{new}(), s, \mathbf{delete } s; \mathbf{delete } r, i, i; \mathbf{delete } r), i, i) : 1 \quad (12)$$

2.3 Central types

An important notion that we will need is that of types that commute for $\otimes$ with any other types, the *central* types. Central types include purely positive types without resources:

$$W ::= 1 \mid W \otimes W' \mid W \oplus W'$$

$\langle (\text{let } x^- = v \text{ in } t)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle^\epsilon$	
$\langle (\text{let } x^+ = t \text{ in } u)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t \mid (x^+.u)^\epsilon \cdot s \mid l \rangle^+$	$\langle v \mid (x^+.t)^\epsilon \cdot s \mid l \rangle^+ \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle^\epsilon$
$\langle (vw)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle v \mid w^\epsilon \cdot s \mid l \rangle^-$	$\langle \lambda x. t \mid v^\epsilon \cdot s \mid l \rangle^- \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle^\epsilon$
$\langle (\pi_i v)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle v \mid \pi_i^\epsilon \cdot s \mid l \rangle^-$	$\langle \langle t_1, t_2 \rangle \mid \pi_i^\epsilon \cdot s \mid l \rangle^- \rightsquigarrow \langle t_i \mid s \mid l \rangle^\epsilon$
$\langle \delta((v, w), (x, y).t)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t[v/x, w/y] \mid s \mid l \rangle^\epsilon$	$\langle \text{new} \mid () \cdot s \mid r_n :: l \rangle^- \rightsquigarrow \langle t_1 r_n \mid s \mid l \rangle^+$
$\langle \delta((), ().t)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t \mid s \mid l \rangle^\epsilon$	$\langle \text{new} \mid () \cdot s \mid \text{Nil} \rangle^- \rightsquigarrow \langle t_2 () \mid s \mid \text{Nil} \rangle^+$
$\langle \delta(t_i v, x_1.t_1, x_2.t_2)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t_i[v/x_i] \mid s \mid l \rangle^\epsilon$	$\langle \text{delete} \mid r_n \cdot s \mid l \rangle^- \rightsquigarrow \langle () \mid s \mid r_n :: l \rangle^+$

Fig. 2. Reduction rules

Indeed we can define by induction the values $\text{swap}_W^A : W \otimes A \multimap A \otimes W$ (corresponding indeed to a natural isomorphism between $A \otimes W$ and $W \otimes A$ in the semantics of section 5):

$$\begin{aligned}
& \text{swap}_I^A \stackrel{\text{def}}{=} \lambda p. \delta(p, (a, i). \delta(i, (). (\star, a))) \\
& \text{swap}_{W_1 \oplus W_2}^A \stackrel{\text{def}}{=} \lambda p. \delta(p, (a, w). \delta(w, \\
& \quad w_1. \text{let } p_1 = \text{swap}_{W_1}^A(a, w_1) \text{ in } \delta(p_1, (w_1, a). (t_1 w_1, a)), \\
& \quad w_2. \text{let } p_2 = \text{swap}_{W_2}^A(a, w_2) \text{ in } \delta(p_2, (w_2, a). (t_2 w_2, a)))) \\
& \text{swap}_{W_1 \otimes W_2}^A \stackrel{\text{def}}{=} \\
& \quad \lambda p. \delta(p, (a, w). \delta(w, (w_1, w_2). \\
& \quad \text{let } p_1 = \text{swap}_{W_2}^{A \otimes W_1}((a, w_1), w_2) \text{ in } \delta(p_1, (w_2, p). \delta(p, (a, w_1). \\
& \quad \text{let } p_2 = \text{swap}_{W_1}^{W_2 \otimes A}((w_2, a), w_1) \text{ in } \delta(p_2, (w_1, p). \delta(p, (w_2, a). ((w_1, w_2), a))))))
\end{aligned}$$

2.4 Operational semantics

We define in fig. 2 a small-step operational semantics for untyped expressions with an abstract machine. We begin by defining a stack that stores the arguments of delayed operations along with their polarity: indices of lazy pair projections, arguments of function applications and continuations of let bindings of positive values. Finally, a command consists of an expression, a stack, an ambient list of resources interpreting the allocation monad and the current polarity.

$$\begin{aligned}
& \text{Stacks } s ::= \star \mid v^\epsilon \cdot s \mid \pi_i^\epsilon \cdot s \mid (x^+.u)^\epsilon \cdot s \\
& \text{Lists } l ::= \text{Nil} \mid r_{n \in \mathbb{N}} :: l \\
& \text{Commands } c ::= \langle t \mid s \mid l \rangle^\epsilon
\end{aligned}$$

We now define reduction rules: let bindings of negative expressions substitute them immediately, following the call-by-name convention, while let bindings of positive expressions push their continuation on the stack to first reduce the expression to a value.

$\star : A \vdash_p A$	$\frac{s : B_\epsilon \vdash_p C \quad \vdash_p v : A}{v^\epsilon \cdot s : B \multimap A \vdash_p C}$	$\frac{s : B_\epsilon \vdash_p C \quad x : A \vdash_p t : B}{(x^+.t)^\epsilon \cdot s : A \vdash_p C}$
$\frac{s : A_\epsilon \vdash_p C}{\pi_1^\epsilon \cdot s : A \& B \vdash_p C}$	$\frac{s : B_\epsilon \vdash_p C}{\pi_2^\epsilon \cdot s : A \& B \vdash_p C}$	

Fig. 3. Stack typing rules

All rules that push arguments on the stack come with their dual rule that consumes the argument on the stack. Pattern match rules reduce instantly as only values can be the scrutinee. Finally, rules for constants **new** and **delete** are the only ones that manipulate the ambient list of resources.

If we interpret our previous program with the list of resources $r_0 :: r_1 :: l$, we obtain in particular the following reduction steps:

$$\begin{aligned}
& \langle \delta(\mathbf{new}(), r. \delta(\mathbf{new}(), s. \mathbf{delete} s; \mathbf{delete} r, i.i; \mathbf{delete} r), i.i) \mid \star \mid r_0 :: r_1 :: l \rangle^+ \\
& \rightsquigarrow^* \langle l_1 r_1 \mid (x^+. \delta(x, s. \mathbf{delete} s; \mathbf{delete} r_0, i.i; \mathbf{delete} r_0)) \cdot \star \mid l \rangle^+ \\
& \rightsquigarrow^* \langle () \mid \star \mid r_0 :: r_1 :: l \rangle^+
\end{aligned}$$

2.5 Properties

We now study properties of $\mathcal{L}$. First, we establish standard properties such as confluence (through determinism), subject reduction and progress.

Theorem 1. *The reduction rules are deterministic: at most one reduction rule can be applied to any command (see section B.2).*

We define in fig. 3 typing judgments for expressions $\vdash_p t : A$ by extending all typing rules given in section 2.2 with the following axiom scheme for resources: $\vdash_p r_n : R$. We then extend those judgements below for closed stacks $s : A \vdash_p B$ which follow sequent calculus rules. Finally, we extend judgements to commands $c : A$, which consist of pairs of judgements $\vdash_p t : B_\epsilon$ and $s : B \vdash_p A$ for $c = \langle t \mid s \mid l \rangle^\epsilon : A$.

We prove the following substitution lemma by induction on the derivation of t :

Lemma 2 (Substitution lemma (SL)). *If $\vdash_p v : A$ and $\Gamma, x : A, \Gamma' \vdash_p t : B$, then $\Gamma, \Gamma' \vdash_p t[v/x] : B$.*

Theorem 3 (Subject reduction). *The reduction rules preserve typing judgements: for any $c_1 : A$ and c_2 , $c_1 \rightsquigarrow c_2$ implies $c_2 : A$. This also holds in the ordered fragment, i.e. for $\vdash_p$ without the exchange rule. (See section B.3.)*

We define final values $v_t ::= () \mid (v, w) \mid l_i v \mid r_n \mid \langle t, u \rangle \mid \lambda x. t \mid \mathbf{new} \mid \mathbf{delete}$. They include all values v such that $\vdash_p v : A_+$.

Theorem 4 (Progress). *A well-typed command reduces if and only if it is not of the shape $\langle v_t \mid \star \mid l \rangle^\epsilon$. This also holds in $\mathcal{O}$. (See section B.4.)*

Together, subject reduction and progress ensures that any well-typed expression $\vdash_p t : A$ reduces either indefinitely or to a final command $\langle v_t \mid \star \mid l \rangle$ with $\vdash_p v_t : A$.

3 Resource-safety properties

In this section, we prove a resource-safety properties for $\mathcal{O}$ pertaining to its ordered character: resources are freed in a last-in-first-out fashion. Then we adapt the proof to prove a weaker property for the whole calculus: resources are freed in random order.

3.1 Ordered case

We only have left functions in $\mathcal{O}$, that is, functions binding the leftmost variable. First note that if we allow right functions in $\mathcal{O}$, then we can interleave a right function between two allocations that allows to write a program that permutes the first two resources:

$$p \stackrel{\text{def}}{=} \delta(\text{new}(), r. \delta(\delta(\text{new}(), s. \underline{\lambda x. \text{delete } x; \text{delete } s, i.i}), f. f r, i. \text{delete } r; i), i.i) \\ \langle p \mid \star \mid r_0 :: r_1 :: l \rangle \rightsquigarrow^* \langle () \mid \star \mid r_1 :: r_0 :: l \rangle$$

In this example that uses, incorrectly, a right function (underlined), the order of r_0 and r_1 are exchanged. The calculus $\mathcal{O}$ ensures that all complete programs act as the identity on the ambient list of resources. We show indeed that for any purely positive type without resource W , executing a closed expression $\vdash t : W$ typed without structural rules leaves the freelist as it has found it initially:

Proposition 5. *For all v such that $\vdash v : W$ and all lists of resources l, l' , if*

$$\langle t \mid \star \mid l \rangle \rightsquigarrow^* \langle v \mid \star \mid l' \rangle$$

then $l' = l$.

To show this, we will define the list of resources of a term $LR(t)$ by tracking resources in contexts of typing rules and show that such lists are preserved by reduction. To reason about resources in case of substitutions, $LR(t[v/x])$ should be equal to the concatenation of $LR(t)$ and $LR(v)$. This restricts contexts to have the shape $\Theta \stackrel{\text{def}}{=} \Gamma; L; \Delta$ with L the list of resources, Γ, Δ contexts of variables and x to be the right-most variable in Γ or the left-most variable in Δ .

Given $\Theta = \Gamma_\Theta; L_\Theta; \Delta_\Theta$ and $\Theta' = \Gamma_{\Theta'}; L_{\Theta'}; \Delta_{\Theta'}$, the expression $\Theta @ \Theta'$ asserts that we are in one of the three following cases to define the concatenation of both contexts:

- $L_\Theta = []$, then $\Theta @ \Theta' \stackrel{\text{def}}{=} \Gamma_\Theta, \Delta_\Theta, \Gamma_{\Theta'}; L_{\Theta'}; \Delta_{\Theta'}$.
- $L_{\Theta'} = []$, then $\Theta @ \Theta' \stackrel{\text{def}}{=} \Gamma_\Theta; L_\Theta; \Delta_\Theta, \Gamma_{\Theta'}, \Delta_{\Theta'}$.
- $\Delta_\Theta = \Gamma_{\Theta'} = \emptyset$, then $\Theta @ \Theta' \stackrel{\text{def}}{=} \Gamma_\Theta; L_\Theta \uplus L_{\Theta'}; \Delta_{\Theta'}$.

We can now type terms with resources with the judgment $\vdash_o$ indexed by such contexts (see fig. 4, omitting types for brevity). Given an ordered expression $\Gamma; L; \Delta \vdash_o t$, we define its list of resources $LR(t) \stackrel{\text{def}}{=} L$. For $\Theta = \Gamma; L; \Delta$, we define the notations $\Theta, x \stackrel{\text{def}}{=} \Gamma; L; \Delta, x$ and $x, \Theta \stackrel{\text{def}}{=} x, \Gamma; L; \Delta$.

Lemma 6. *If $\Gamma \vdash t : A$, then $\Gamma; ; \vdash_o t : A$ (see section C.1).*

We can then accept substitutions of expressions that preserve well-formed contexts. The following left and right substitution lemmas cover substitutions encountered in the operational semantics.

$\frac{}{; []; x \vdash_o x}$	$\frac{}{; []; \vdash_o ()}$	$\frac{}{; []; \vdash_o \text{new}}$	$\frac{}{; []; \vdash_o \text{delete}}$	$\frac{}{; [r_n]; \vdash_o r_n}$
$\frac{\Gamma; []; x, \Delta \vdash_o x}{\Gamma, x; []; \Delta \vdash_o x}$	$\frac{\Gamma, x; []; \Delta \vdash_o x}{\Gamma; []; x, \Delta \vdash_o x}$			
$\frac{\Theta \vdash_o v}{\Theta \vdash_o l_i v}$	$\frac{\Theta \vdash_o v}{\Theta \vdash_o \pi_i v}$	$\frac{\Theta \vdash_o t \quad \Theta \vdash_o u}{\Theta \vdash_o \langle t, u \rangle}$	$\frac{x, \Theta \vdash_o t}{\Theta \vdash_o \lambda x. t}$	$\frac{\Theta \vdash_o v \quad \Theta' \vdash_o w}{\Theta @ \Theta' \vdash_o (v, w)}$
$\frac{\Theta, x \vdash_o u \quad \Theta' \vdash_o t}{\Theta @ \Theta' \vdash_o \text{let } x = t \text{ in } u}$	$\frac{\Theta, x, y @ \Theta'' \vdash_o t \quad \Theta' \vdash_o v}{\Theta @ \Theta' @ \Theta'' \vdash_o \delta(v, (x, y).t)}$	$\frac{\Theta @ \Theta'' \vdash_o t \quad \Theta' \vdash_o v}{\Theta @ \Theta' @ \Theta'' \vdash_o \delta(v, ().t)}$		
$\frac{\Theta, x @ \Theta'' \vdash_o t \quad \Theta, y @ \Theta'' \vdash_o u \quad \Theta' \vdash_o v}{\Theta @ \Theta' @ \Theta'' \vdash_o \delta(v, x.t, y.u)}$	$\frac{\Theta \vdash_o w \quad \Theta' \vdash_o v}{\Theta @ \Theta' \vdash_o vw}$			

Fig. 4. Typing rules of ordered expressions with resources

$\frac{}{[] \vdash_o^S \star}$	$\frac{; L_v; \vdash_o v \quad L_s \vdash_o^S s}{L_s \vdash L_v \vdash_o^S v \cdot s}$	$\frac{; L_t; x \vdash_o t \quad L_s \vdash_o^S s}{L_s \vdash L_t \vdash_o^S (x^+.t) \cdot s}$
	$\frac{; L_t; \vdash_o t \quad L_s \vdash_o^S s}{L_s \vdash L_t \vdash_l \vdash_o^C \langle t \mid s \mid l \rangle}$	

Fig. 5. Typing rules of ordered stacks and commands

Lemma 7. (Left SL) *If $; L_t; \vdash_o t$ and $\Gamma, x; L_u; \Delta \vdash_o u$, then $\Gamma; L_t \vdash L_u; \Delta \vdash_o u[t/x]$.*
(Right SL) *If $; L_t; \vdash_o t$ and $\Gamma; L_u; x, \Delta \vdash_o u$, then $\Gamma; L_u \vdash L_t; \Delta \vdash_o u[t/x]$ (see section C.2).*

We then extend $\vdash_o$ for stacks and commands, with only resources in their contexts. Resources from stacks remain to the left of resources from expressions in the context of commands:

Theorem 8. *Reducing an ordered command results in an ordered command with the same list of resources, i.e. for all c_1, c_2, M such that $L \vdash_o^C c_1$ and $c_1 \rightsquigarrow^* c_2$ one has $L \vdash_o^C c_2$ (see section C.3).*

We can now prove proposition 5:

Proof. For any expression of a central type $\vdash t : W$ in $\mathcal{O}$, by subject reduction and progress we have that $\langle t \mid \star \mid l \rangle$ either reduces indefinitely, or there exists a final value $\vdash v_t : W$ such that $\langle t \mid \star \mid l \rangle \rightsquigarrow^* \langle v \mid \star \mid l' \rangle$.

We then prove by induction on derivations that typed expressions without structural rules are ordered: they do not include resources, so concatenated contexts are always well-formed. So both t and v are ordered and without values. Because reduction rules preserve the list of resources, we have $l = l'$.

3.2 Linear case

If we consider expressions in $\mathcal{L}$, the property is relaxed the property for $\vdash t : W$:

Theorem 9. *For all v, l, l' such that*

$$\langle t \mid \star \mid l \rangle \rightsquigarrow^* \langle v \mid \star \mid l' \rangle$$

there exists a permutation σ of lists of resources such that $l' = \sigma(l)$.

Proof. This is proved similarly to proposition 5, by forgetting the order of variables and resources in the predicate $\Gamma; L; \Delta \vdash_o t$. This simplifies context concatenation by removing preconditions and requires a single substitution lemma. The proof is detailed in section C.4.

4 Extending the calculus of allocations with destructors and exceptions

We now define an extension of $\mathcal{L}$ with exceptions and affine values that we name $\mathcal{O}_{\mathcal{E}, \text{move}}$. Its semantics is given by translation in $\mathcal{L}$: exceptions are propagated as errors, removing variables from the context with their associated destructor. Hence, $\mathcal{L}$ serves as a meta-language in which the ambient allocation monad remains implicit, and which give resource-safety properties for $\mathcal{O}_{\mathcal{E}, \text{move}}$.

4.1 Grammar

The grammar of terms of $\mathcal{O}_{\mathcal{E}, \text{move}}$ extends that for $\mathcal{L}$, with the following terms:

- “**drop**”, a value that drops its argument.
- “**move**(x, y) in t ”, an expression that explicitly exchanges variables.
- “**raise**”, a value that inhabits any type with a given exception.
- “**try** $x \Leftarrow t$ in u **unless** $e \Rightarrow u'$ ”, an expression that catches exceptions occurring in t based on [4].

In the remainder, we leave polarity annotations implicit because they can always be inferred from the type.

4.2 Type system

The grammar of types is unchanged:

$$\text{Types } A, B : \begin{cases} \text{Positive types} & P, Q ::= R \mid 1 \mid A \otimes B \mid A \oplus B \\ \text{Negative types} & N, M ::= B \multimap A \mid A \& B \end{cases}$$

However, the interpretation of types is changed: positive types are assigned a chosen destructor. Implicitly, positive types are interpreted as objects in the slice category over

$\frac{}{x : A \vdash x : A} \text{var}$	$\frac{}{\vdash \mathbf{drop} : 1 \multimap A} \text{drop}$
$\frac{\Gamma, x : A, y : B, \Gamma' \vdash t : C}{\Gamma, y : B, x : A, \Gamma' \vdash \mathbf{move}(x, y) \text{ in } t : C} \text{move}$	
$\frac{}{\vdash \mathbf{new} : R \multimap 1} \text{new}$	$\frac{\Delta \vdash t : A \quad \Gamma, x : A \vdash u : B}{\Gamma, \Delta \vdash \text{let } x = t \text{ in } u : B} \text{let}$
$\frac{}{\vdash \mathbf{raise} : A \multimap E} \text{raise}$	$\frac{\Delta \vdash t : P \quad \Gamma, x : P \vdash u : A \quad \Gamma, e : E \vdash u' : A}{\Gamma, \Delta \vdash \mathbf{try } x \Leftarrow t \text{ in } u \text{ unless } e \Rightarrow u' : A} \text{try}$
$\frac{\Gamma \vdash v : A \quad \Delta \vdash w : B}{\Gamma, \Delta \vdash (v, w) : A \otimes B} \otimes_i$	$\frac{\Delta \vdash v : A \otimes B \quad \Gamma, x : A, y : B, \Gamma' \vdash t : C}{\Gamma, \Delta, \Gamma' \vdash \delta(v, (x, y).t) : C} \otimes_e$
$\frac{}{\vdash () : 1} 1_i$	$\frac{\Delta \vdash v : 1 \quad \Gamma, \Gamma' \vdash t : A}{\Gamma, \Delta, \Gamma' \vdash \delta(v, ().t) : A} 1_e$
$\frac{\Gamma \vdash v : A}{\Gamma \vdash i_1 v : A \oplus B} \oplus_{i1}$	$\frac{\Gamma \vdash v : B}{\Gamma \vdash i_2 v : A \oplus B} \oplus_{i2}$
$\frac{\Delta \vdash v : A \oplus B \quad \Gamma, x : A, \Gamma' \vdash t : C \quad \Gamma, y : B, \Gamma' \vdash u : C}{\Gamma, \Delta, \Gamma' \vdash \delta(v, x.t, y.u) : C} \oplus_e$	
$\frac{x : A, \Gamma \vdash t : B}{\Gamma \vdash \lambda x.t : B \multimap A} \multimap_i$	$\frac{\Gamma \vdash w : A \quad \Delta \vdash v : B \multimap A}{\Gamma, \Delta \vdash vw : B} \multimap_e$
$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \& B} \&_i$	$\frac{\Gamma \vdash v : A \& B}{\Gamma \vdash \pi_1 v : A} \&_{e1} \quad \frac{\Gamma \vdash v : A \& B}{\Gamma \vdash \pi_2 v : B} \&_{e2}$

Fig. 6. Typing rules for $\mathcal{O}_{\mathcal{E}, \text{move}}$

TI , which is reflected in our ordered adaptation of the interpretation of linear call-by-push-value with a resource modality [10, Theorem 19].

The previous primitive $\mathbf{new} : R \oplus 1 \multimap 1$ with explicit errors is replaced by $\mathbf{new} : R \multimap 1$: the function may throw an exception when called. Exceptions must not exchange resources during stack unwinding, so the calculus is parametrized by a definable central type E and a definable value $\vdash \text{new_fail} : E$ thrown whenever an allocation fails (for instance, $E = 1$ and $\text{new_fail} = ()$). We hence rely on swap expressions in $\mathcal{O}_{\mathcal{E}, \text{move}}$, defined as in $\mathcal{L}$.

$\mathbf{move}$ exchanges variables, but is not presented as a structural rule as it performs an effect. Indeed, $A \otimes B$ and $B \otimes A$ fix different orders of destruction of A and B for exception propagation, thus they can be distinguished observationally. In particular, $\mathbf{move}$ cannot give rise to a *contextual isomorphism* [30] between $A \otimes B$ and $B \otimes A$. The reason is that although $\mathbf{move}(x, y)$ and $\mathbf{move}(y, x)$ are inverse of each other, a contextual isomorphism also requires $\mathbf{move}$ to be pure [30].

$$A \otimes B \dashv\vdash B \otimes A \text{ but } A \otimes B \not\cong B \otimes A \quad (13)$$

We name the fragment without $\mathbf{move}$ $\mathcal{O}_{\mathcal{E}}$.

For example, the following term allocates three resources then frees them in reverse order. If the last allocation fails, then the raised exception will free the first two resources.

This example is similar to the one in $\mathcal{O}$ (see eq. (12)), which needed to deal with allocation failures explicitly.

let $r = \mathbf{new}()$ in let $s = \mathbf{new}()$ in let $t = \mathbf{new}()$ in **drop** t ; **drop** s ; **drop** r

4.3 Translation into $\mathcal{L}$

We follow the ordered call-by-push-value translation obtained from the resource modality between values with destructors and throwable computations, with shifts $\uparrow A \stackrel{\text{def}}{=} A \oplus E$ and $\downarrow A \stackrel{\text{def}}{=} A \& 1$. Indeed, recall that $- \& TI$ (i.e. $A \& 1$ at the level of types) gives rise to the right adjoint to the forgetful functor of the slice category (eq. (11)). Each type A has a positive interpretation A^+ equipped with a destructor written drop_A and a negative interpretation A^- .

$$\begin{array}{ll}
1^+ \stackrel{\text{def}}{=} 1 & \text{drop}_1 \stackrel{\text{def}}{=} \lambda v.v \\
R^+ \stackrel{\text{def}}{=} R & \text{drop}_R \stackrel{\text{def}}{=} \lambda r. \mathbf{delete} \, r \\
(A \otimes B)^+ \stackrel{\text{def}}{=} A^+ \otimes B^+ & \text{drop}_{(A \otimes B)} \stackrel{\text{def}}{=} \lambda p. \delta(p, (a, b). \text{drop}_B \, b; \text{drop}_A \, a) \\
(A \oplus B)^+ \stackrel{\text{def}}{=} A^+ \oplus B^+ & \text{drop}_{(A \oplus B)} \stackrel{\text{def}}{=} \lambda s. \delta(s, a. \text{drop}_A \, a, b. \text{drop}_B \, b) \\
N^+ \stackrel{\text{def}}{=} \downarrow A^- & \text{drop}_N \stackrel{\text{def}}{=} \lambda a. \pi_2 a \\
(B \multimap A)^- \stackrel{\text{def}}{=} B^- \multimap A^+ & \\
(A \& B)^- \stackrel{\text{def}}{=} A^- \& B^- & \star^+ \stackrel{\text{def}}{=} \star \\
P^- \stackrel{\text{def}}{=} \uparrow A^+ & (\Gamma, x : A)^+ \stackrel{\text{def}}{=} \Gamma^+, x : A^+
\end{array}$$

Note that for any purely positive type (hence also E), we have $W^+ = W$. We define three expressions which we use in the translation:

- $\Gamma^+ \vdash \text{drop_ctx}_\Gamma : 1$ by induction on Γ , which drops each variable from right to left.

$$\begin{aligned}
\text{drop_ctx}_\star &\stackrel{\text{def}}{=} () \\
\text{drop_ctx}_{\Gamma, x : A} &\stackrel{\text{def}}{=} \text{drop}_A \, x; \text{drop_ctx}_\Gamma
\end{aligned}$$

- $\Gamma^+, e : E \vdash \text{unwind}_\Gamma(e) : E$ by induction on Γ , which drops the context and returns e .

$$\begin{aligned}
\text{unwind}_\star(e) &\stackrel{\text{def}}{=} e \\
\text{unwind}_{\Gamma, x : A}(e) &\stackrel{\text{def}}{=} \text{let } p = \text{swap}_E^A(x, e) \text{ in } \delta(p, (e, x). \text{drop}_A \, x; \text{unwind}_\Gamma(e))
\end{aligned}$$

- $\Gamma^+, e : E \vdash \text{raise}_\Gamma^A(e) : A^-$ by induction on A , which drops the context and inhabits A^- with e .

$$\begin{aligned}
\text{raise}_\Gamma^{C \multimap B}(e) &\stackrel{\text{def}}{=} \lambda b. \text{raise}_{b : B, \Gamma}^C(e) \\
\text{raise}_\Gamma^{B \& C}(e) &\stackrel{\text{def}}{=} \langle \text{raise}_\Gamma^B(e), \text{raise}_\Gamma^C(e) \rangle \\
\text{raise}_\Gamma^P(e) &\stackrel{\text{def}}{=} \text{let } e' = \text{unwind}_\Gamma(e) \text{ in } \iota_2 e'
\end{aligned}$$

We then translate $\mathcal{O}_{\mathcal{E},\text{move}}$ derivations with two functions noted $\llbracket - \rrbracket$ defined by mutual induction on derivations of the two kinds of judgements:

- $\mathcal{O}_{\mathcal{E},\text{move}}$ values of type $\Gamma \vdash A$ are translated to $\mathcal{L}$ values of type $\Gamma^+ \vdash A^+$.
- $\mathcal{O}_{\mathcal{E},\text{move}}$ expressions of type $\Gamma \vdash A$ are translated to $\mathcal{L}$ values of type $\Gamma^+ \vdash \Downarrow A^-$.

We add explicit coercions to treat positive values as expressions with the notation $\text{coerc}(v)$. The monad strength for $T\mathcal{E}$ can be observed when translating let bindings of expressions, where the context of the continuation is dropped in case of an error.

$$\begin{aligned}
ax : \llbracket x \vdash x : A \rrbracket &\stackrel{\text{def}}{=} x \\
\text{drop} : \llbracket \vdash \text{drop} : 1 \multimap A \rrbracket &\stackrel{\text{def}}{=} \langle \text{drop}_A, () \rangle \\
\text{coerc} : \llbracket \Gamma \vdash \text{coerc}(v) : P \rrbracket &\stackrel{\text{def}}{=} \langle \iota_1 \llbracket v \rrbracket, \text{drop_ctx}_\Gamma \rangle \\
\text{move} : \llbracket \Gamma, x : A, y : B, \Delta \vdash \text{move}(x, y) \text{ in } t : C \rrbracket &\stackrel{\text{def}}{=} \llbracket t \rrbracket \\
\text{new} : \llbracket \vdash \text{new} : Rd \multimap 1 \rrbracket &\stackrel{\text{def}}{=} \text{let } x = \text{new}() \text{ in } \delta(x, r.\iota_1 r, i.i; \iota_2 \text{new_fail}) \\
\text{let}_v : \llbracket \Gamma, \Delta \vdash \text{let } x = (v : A) \text{ in } t : B \rrbracket &\stackrel{\text{def}}{=} \text{let } x = \llbracket v \rrbracket \text{ in } \llbracket t \rrbracket \\
\text{let}_t : \llbracket \Gamma, \Delta \vdash \text{let } x = (t : P) \text{ in } u : A \rrbracket &\stackrel{\text{def}}{=} \text{let } s = \pi_1 \llbracket t \rrbracket \text{ in } \delta(s, x.\llbracket u \rrbracket, e.\text{raise}_\Gamma^{A \& I}(e)) \\
\text{raise} : \llbracket \vdash \text{raise} : A \multimap E \rrbracket &\stackrel{\text{def}}{=} \langle \lambda e. \text{raise}_\star^A(e), () \rangle \\
\text{try} : \llbracket \Gamma, \Delta \vdash \text{try } x \Leftarrow (t : P) \text{ in } u \text{ unless } e \Rightarrow u' : B \rrbracket &\stackrel{\text{def}}{=} \text{let } s = \pi_1 \llbracket t \rrbracket \text{ in } \delta(s, x.\llbracket u \rrbracket, e.\llbracket u' \rrbracket) \\
\otimes_i : \llbracket \Gamma, \Delta \vdash (v, w) : A \otimes B \rrbracket &\stackrel{\text{def}}{=} (\llbracket v \rrbracket, \llbracket w \rrbracket) \\
\otimes_e : \llbracket \Gamma, \Delta, \Gamma' \vdash \delta(v, (x, y).t) : C \rrbracket &\stackrel{\text{def}}{=} \delta(\llbracket v \rrbracket, (x, y).\llbracket t \rrbracket) \\
1_i : \llbracket \vdash () : 1 \rrbracket &\stackrel{\text{def}}{=} () \\
1_e : \llbracket \Gamma, \Delta, \Gamma' \vdash \delta(v, ().t) : C \rrbracket &\stackrel{\text{def}}{=} \delta(\llbracket v \rrbracket, ().\llbracket t \rrbracket) \\
\oplus_{i1} : \llbracket \Gamma \vdash \iota_1 v : A \oplus B \rrbracket &\stackrel{\text{def}}{=} \iota_1 \llbracket v \rrbracket \\
\oplus_{i2} : \llbracket \Gamma \vdash \iota_2 v : A \oplus B \rrbracket &\stackrel{\text{def}}{=} \iota_2 \llbracket v \rrbracket \\
\oplus_e : \llbracket \Gamma, \Delta, \Gamma' \vdash \delta(v, x.t, y.u) : C \rrbracket &\stackrel{\text{def}}{=} \delta(\llbracket v \rrbracket, x.\llbracket t \rrbracket, y.\llbracket u \rrbracket) \\
\multimap_i : \llbracket \Gamma \vdash \lambda x. t : B \multimap A \rrbracket &\stackrel{\text{def}}{=} \langle \lambda x. \pi_1 \llbracket t \rrbracket, \text{drop_ctx}_\Gamma \rangle \\
\multimap_e : \llbracket \Gamma, \Delta \vdash v w : B \rrbracket &\stackrel{\text{def}}{=} \langle (\pi_1 \llbracket v \rrbracket) \llbracket w \rrbracket, \text{drop_ctx}_{\Gamma, \Delta} \rangle \\
\&_i : \llbracket \Gamma \vdash \langle t, u \rangle : A \& B \rrbracket &\stackrel{\text{def}}{=} \langle \langle \pi_1 \llbracket t \rrbracket, \pi_1 \llbracket u \rrbracket \rangle, \text{drop_ctx}_\Gamma \rangle \\
\&_{e1} : \llbracket \Gamma \vdash \pi_1 v : A \rrbracket &\stackrel{\text{def}}{=} \langle \pi_1 \pi_1 \llbracket v \rrbracket, \text{drop_ctx}_\Gamma \rangle \\
\&_{e2} : \llbracket \Gamma \vdash \pi_2 v : B \rrbracket &\stackrel{\text{def}}{=} \langle \pi_2 \pi_1 \llbracket v \rrbracket, \text{drop_ctx}_\Gamma \rangle
\end{aligned}$$

Note that expressions $t : P$ get a destructor that is immediately discarded and are evaluated eagerly in let bindings. Hence, they are treated like expressions of type $A \oplus E$. We thus define the operational semantics of an affine value v to be that of $\llbracket v \rrbracket$, and the operational semantics of an affine expression t to be that of $\pi_1 \llbracket t \rrbracket$.

Theorem 10. *For any expression t of a central type W in $\mathcal{O}_{\mathcal{E},\text{move}}$, we have:*

$$\forall v, l, l'. \langle \pi_1 \llbracket t \rrbracket \mid \star \mid l \rangle \rightsquigarrow^* \langle v \mid \star \mid l' \rangle \Rightarrow \exists \sigma. l' = \sigma(l)$$

Moreover, if the expression is in $\mathcal{O}_{\mathcal{E}}$, we have the following stronger property:

$$\forall v, l, l'. \langle \pi_1 \llbracket t \rrbracket \mid \star \mid l \rangle \rightsquigarrow^* \langle v \mid \star \mid l' \rangle \Rightarrow l' = l$$

Proof. The translation defined below is well-typed, and use $\mathcal{L}$ structural rules only for **move** (see section D.1), so the $\mathcal{O}_{\mathcal{E}}$ fragment is translated in $\mathcal{O}$. Because central types are translated to the identity, $\Downarrow W^- = (W \oplus E) \& I$, hence $\pi_1 \llbracket t \rrbracket$ is a linear expression of type $W \oplus E$. E is a central type, so $W \oplus E$ as well. Hence, the resources safety properties of $\mathcal{L}$ apply to $\pi_1 \llbracket t \rrbracket$, as well as those of $\mathcal{O}$ if t is in $\mathcal{O}_{\mathcal{E}}$. Those properties are exactly the properties we needed to demonstrate.

5 A presheaf model of $\mathcal{O}$

The ordered calculus $\mathcal{O}$ shows some peculiarities compared to usual formulations of ordered logic: it has a left arrow and no right arrow, and a right let binding without left let binding. We find it illustrative to give a simple concrete model showing that these peculiarities are natural and arise by construction upon consideration of the strengths for the allocation monad.

We interpret the calculus $\mathcal{O}$ in the presheaf category $\mathcal{L} \stackrel{\text{def}}{=} \text{Set}^{[\mathbf{R}]}$, with $[\mathbf{R}]$ the set of lists of natural numbers considered as a discrete category. We note $\#$ the non-symmetric monoidal product on $[\mathbf{R}]$ obtained by concatenating two lists of resources. Its Day convolution [11] yields the multiplicative fragment. We define $R \in \text{Set}^{[\mathbf{R}]}$ to be the indicator function for singleton lists. As a presheaf category, $\mathcal{L}$ has products and coproducts. The initial algebra of $I \oplus (R \otimes -)$ interpreting lists of resources is the terminal object $\top$. Concretely, we have the following constructions in $\mathcal{L}$:

- The monoidal product $(A \otimes B)(l) \stackrel{\text{def}}{=} \exists(l_1, l_2 \mid l_1 \# l_2 = l). A(l_1) \times B(l_2)$.
- The unit $I(\top) \stackrel{\text{def}}{=} \{\emptyset\}$, $I(l) \stackrel{\text{def}}{=} \emptyset$ is the indicator function for the empty list.
- The right arrow $(A \multimap B)(l_1) \stackrel{\text{def}}{=} \forall l_2. A(l_2) \rightarrow B(l_1 \# l_2)$.
- The left arrow $(B \multimap A)(l_2) \stackrel{\text{def}}{=} \forall l_1. A(l_1) \rightarrow B(l_1 \# l_2)$.
- The type of resources $R([n]) \stackrel{\text{def}}{=} \{n\}$, $R(l) \stackrel{\text{def}}{=} \emptyset$.
- The type of lists of resources $[R](l) \stackrel{\text{def}}{=} \{l\}$.
- The product $(A \& B)(l) \stackrel{\text{def}}{=} A(l) \times B(l)$.
- The coproduct $(A \oplus B)(l) \stackrel{\text{def}}{=} A(l) + B(l)$.

We obtain the allocation monad from the adjunction corresponding to the right closed structure

$$F \stackrel{\text{def}}{=} (- \otimes [R]) \dashv [R] \multimap -$$

The tensor associator gives a natural isomorphism

$$A \otimes FB \simeq F(A \otimes B). \quad (14)$$

We now have all the ingredients to interpret the calculus $\mathcal{O}$. We interpret expressions typed with $\Gamma; [r_1 \cdots r_n]; \Delta \vdash_o A$ as (oblique) morphisms in $\mathcal{L}(F(\Gamma \otimes R^n \otimes \Delta), A)$, following [10] adapted to the ordered case. In particular, the isomorphism (eq. (14)) gives rise to a strong adjunction in the sense of [10] albeit in an ordered setting.

Interpreting the left function and the right let binding uses the left strength of the allocation monad. We cannot interpret a right function or left let binding for $\mathcal{O}$, as the allocation monad does not have a right strength without symmetry.

We defer the complete definition of the interpretation to appendix E.

Lemma 11 (Substitution lemma). *For all $\Theta'' \vdash_o v : A$ and $\Theta, x : A @ \Theta' \vdash_o t : B$, one has:*

$$\llbracket t[v/x] \rrbracket = (id_{\Theta^+} \otimes \llbracket v \rrbracket \otimes id_{\Theta'^+}); \llbracket t \rrbracket$$

(see section E.2).

Theorem 12 (Soundness of the interpretation). *For all L, c, c' such that $L \vdash_o c$ and $L \vdash_o c'$, if $c \rightsquigarrow c'$ then $\llbracket c \rrbracket = \llbracket c' \rrbracket$ (see section E.3).*

5.1 Exceptions and destructors

Exceptions need to be chosen among central objects in order to define a strength for the exception monad. Semantically, we ask for objects that commute with all other objects A in a coherent way. Those are the objects in the *Drinfeld center* of $\mathcal{L}$.

Proposition 13. *Given an object E , the three following properties are equivalent:*

- E is in the Drinfeld center of $\mathcal{L}$.
- E is resource-free: for all non-empty list l , $E(l) = \emptyset$.
- E has a trivial destructor: there is a morphism $\mathcal{L}(E, I)$.

See the proof in section E.4. By induction, central types in $\mathcal{L}$ are resource-free, therefore they belong to the Drinfeld center of $\mathcal{L}$.

Proposition 14. *Every object in $\mathcal{L}$ has a (unique) destructor, but $\mathcal{L}$ is not a model of affine logic in the sense of having an isomorphism $I \simeq \top$ (cf. [8, §3.2]).*

Proof. Recall that $[R] = \top$ is the terminal object. Destructors for A are objects in $(A, \delta) \in \mathcal{L}/TI$; i.e. $\delta \in \mathcal{L}(A, TI) \simeq \mathcal{L}(A, [R] \multimap [R]) \simeq \mathcal{L}(A \otimes \top, \top)$ and no other condition. By the universal property of $\top$ there exists a unique such morphism. On the other hand, only central objects have a trivial destructor $\mathcal{L}(A, I)$, which is not the case for $[R] = \top$; indeed $[R](l) = \{l\}$ so there is no morphism $[R] \rightarrow I$.

Given that any Γ has a unique destructor, one has $\mathcal{L}/TI \cong \mathcal{L}$. In particular, the monad $T\mathcal{E}$ is left strong.

Proposition 15. *Let E a central object. The monad $T\mathcal{E}$ where $\mathcal{E} = - \oplus E$ has a left strength $(\Gamma \otimes T\mathcal{E}A \rightarrow T\mathcal{E}(\Gamma \otimes A))$.*

Thus, this model is affine in the sense of the calculus with destructors $\mathcal{O}_{\mathcal{E}}$, without being a model of affine logic in the usual sense of semi-cartesian monoidal categories.

6 Discussion

After this formal development, we find it useful to replace our results in the context of the theory and practice of programming languages.

6.1 Integrating linear types and error handling

We modelled a global allocator with a linear state monad to study notions of resource-safety for linear calculi: every allocated resource in $\mathcal{L}$ and $\mathcal{O}_{\mathcal{E}, \text{move}}$ is freed exactly once, even in case of errors in the program. This result strengthens when the exchange rule (6) is not used: in $\mathcal{O}$ and $\mathcal{O}_{\mathcal{E}}$, the resources are released in a LIFO order. A relationship between ordered logic and stack-like allocation was proposed previously [44, 45, 55, 42]. In this context, our results suggest to consider a variant of ordered logic that only has one arrow type and whose composition (let) is restricted to the opposite side of abstraction (λ)—where in particular λ cannot express let. The modelling of “move” as an exchange rule that performs a side-effect is also novel.

Conceptually close to stack allocation, LISP’s higher-order combinator `unwind-protect` that executes a clean-up function upon normal or exceptional return of its argument (like its variants found in modern academic programming languages such as OCaml and Haskell) has the same constraint of deallocation in LIFO order.

“Linear types”, on the other hand, promised to let us manipulate resources as first-class values (see [2, 1] among others). However, extensive practical experiments with linear types stumbled on the problematic interaction of linearity with error handling, another important aspect with resources that are acquired in a program. For instance, Cyclone permitted resource leaks in the name of “*flexibility and usability*” [50, §3.5] (necessitating back-up collection mechanisms), whereas Tov and Pucella mentioned the issue of combining linearity and control effects as a motivation for their practical linear type system to be affine (and thus not meant to represent resources as first-class values) [54, 52]. Tov and Pucella [53] proposed separately a system that offers both linearity and control effects (*e.g. checked exceptions*) with an elaborate type-and-effect system that enforces a mutual exclusion principle matching the one described by the notion of strength 8 relative to an affine resource modality (explained in section 1). However, we did not find results regarding the usability of such a system in the literature. Lastly, affine session types [38] do mix linearity and exceptions in our sense. They guarantee a safety property after an exception (cancellation) by propagating cancellation across communication channels.

Explicit error handling with an exception (or error) monad encounters similar obstacles as exceptions, which we can see arising from the lack of unrestricted monadic strength. Concretely, for each location where one would like to use a monadic binding (4), one needs to implement by hand a repetitive description of whichever clean-up functions should be called in case of error propagation—which is sufficiently tedious to have given rise to specific programming patterns in C using `goto` (see appendix A). These lessons about the limitations of linear types for practical purposes were soon forgotten, and recent linear type systems purporting to implement first-class resources in academic languages [6, 40] omit to explore or mention these obstacles.

C++ destructors and its “move” operation, which we set out to model in this paper with $\mathcal{O}$ and $\mathcal{O}_{\mathcal{E}}$, realise a shift in viewpoint whereby clean-up functions are deduced from the types. To begin with, interpreting resource types in the slice category over some TI , *i.e.* as types provided with some chosen effectful weakening map $A \rightarrow TI$, suffices to provide a notion of strength for the exception (or error) monad. Concretely this

means that we can both program with an error monad, and give a meaning to exceptions. Surprisingly, our model presents facets of all three of affine, linear and ordered logics:

$$\frac{\frac{\textit{affine} \text{ in terms of expressiveness and of available control effects}}{\textit{linear} \text{ in terms of resources as values and of resource-safety}}}{\textit{ordered} \text{ in terms of proof-relevance and of type isomorphisms}}$$

Given that the logical expressiveness is that of affine logic (including the exchange rule), without sacrificing the linear character of values (which are correctly released), we do not have to choose between linearity and control effects. This perspective suggests that it should be possible to adapt practical works on affine typing in order to integrate first-class resources in functional programming languages [39].

6.2 A model of C++/Rust-style resource management and its interpretation

This model of C++/Rust-style resource management is at the intersection of linear logic and the theory of effects. Various less idealised models of ownership in Rust have been developed to prove the safety and functional properties of Rust programs, such as with separation logic [25], with a dedicated functional logic [12], or by translation [23] into a purely functional language. These recent works explicitly do not model exceptions with destructor calls, and they do not develop a specific understanding of destructors in relationship with linearity.

To our knowledge, ours is the first model to reproduce various phenomena seen with C++/Rust’s resource management such as: the propagation of errors involving the release of resources in scope as part of monadic binding; destructors having to never fail; resources being first-class values that can be passed, returned, or stored in algebraic data types and closures, and for which “moving” resources (identified with a kind of exchange rule, albeit performing a side-effect) is responsible for altering the order of destruction; and indeed closures themselves possibly being resources, leading to the existence of different types of closures in C++ and Rust.

One very interesting aspect of $\mathcal{O}_{\mathcal{E}, \text{move}}$ is that exchange and weakening are interpreted by morphisms that perform effects. For instance, $A \otimes B$ and $B \otimes A$ are not contextually isomorphic in the language with destructors because they can be distinguished observationally from the order in which the destructors of A and B are executed. So we can have a logical equivalence in the form of inverse morphisms:

$$A \otimes B \rightleftarrows B \otimes A \tag{15}$$

without the two types being isomorphic. A contextual notion of isomorphism indeed requires the two inverse maps (15) to be pure [30], which they are not in our model. This represents a radical departure from symmetric premonoidal categories [46].

We usually see the Curry-Howard correspondence at work when algebraic and logical structures inspire new programming language features. Could the concept of destructors have appeared out of theoretical consideration or practical experiments starting from linear logic? As it turns out, the interpretation of resource types with destructors as ordered logic formulæ almost arose on several occasions, such as when Baker suggested

that C++ constructors and destructors could fit within his linear language, in rarely-cited essays that anticipated C++ move semantics [2, 1], and when Gan, Tov and Morrisett approached substructural types using type classes `dup` and `drop` [16]. But we find it remarkable that the Curry-Howard correspondence actually worked this time in the converse direction, with C++ destructors and move semantics arising from practical consideration over more than 20 years [26, 49, 22], and the use of the slice category in this context being motivated by empirically-observed phenomena with C++ resources [9].

6.3 Conclusion and future work

Our hope is that this newfound understanding of resources in programming will be inspire improvements to programming languages and designs for better ones. It could also help proving more properties of programs with formal methods, such as methods based on translations into pure functional programs. The functional correctness of some (e.g. Rust) programs indeed depends on the linearity properties enforced by the language, for instance when they use the `typestate` programming pattern.

The Curry-Howard correspondence is sometimes stated as a more technical result relating logic, categorical structures and programming language models, as in the Curry-Howard-Lambek correspondence for linear call-by-push-value [10]. Though not essential to the results of this paper, it is now clear that extending the Curry-Howard-Lambek correspondence to ordered call-by-push-value would be useful as a metatheory to study resources. Removing the symmetry in monoidal categories and the exchange rule in logic can create technical issues—we have made some progress in this work already by understanding that λ and `let` bindings work on opposite sides, and that being merely left-closed or right-closed but not both is already interesting and useful.

Lastly, much remains to be understood in an idealised and principled manner within this framework, the most obvious ones being:

- **Copyable types** Linear languages, including with C++ and Rust, mix in practice resource types and unconstrained types. The interaction of the resource modality arising from the slice construction with the resource modality $!$ which controls freely copyable and erasable types, should therefore be explored. Interesting questions also arise from the necessity of kind polymorphism in such languages.

As a starting point, we suggest to see an unrestricted type as a specific kind of resource type whose destructor, obtained from the given weakening, is pure. More precisely, given a symmetric monoidal closed category $\mathcal{C}$ with finite products, a strong monad T , and a resource modality $!$, any unrestricted type appears as a resource type as follows:

Proposition 16. *There is a distributive law for $!$ over the comonad $\mathcal{D} = - \& T I$, which lifts $!$ to a comonad $!_R$ on $\mathcal{C}_{/T I}$. Moreover, there is a full and faithful functor from the category of coalgebras $\mathcal{C}^!$ to the category of coalgebras $\mathcal{C}_{/T I}^{!_R}$ that sends a $!$ -algebra $(A, \tau : A \rightarrow !A)$ to a $!_R$ -algebra whose underlying object in $\mathcal{C}_{/T I}$ is $A_R = (A, w : A \rightarrow T I)$ where w is obtained from the weakening $!A \rightarrow I$. In particular, A_R commutes with all objects $B \in \mathcal{C}_{/T I}$: $A_R \otimes B$ is (contextually) isomorphic to $B \otimes A_R$ in the sense that the two inverse maps (15) for A_R and B are pure.*

- **Borrowing** We have modelled resources that can only be allocated and deallocated, which lets us to observe the effect of deallocations on the final state of the program. More questions arise when resources can be used between their allocation and their deallocation. Operations that do not consume the resource can be implemented by returning it [2, 1]. Borrowing [18, 15], inspired by regions [51], and refined into unique borrowing in Rust, was introduced as a more usable and expressive alternative to this linear threading of values. As we already mentioned, various works account for borrowing using different approaches [25, 12, 23], but it remains to be seen whether and how borrowing may be derived in a principled way from a concept of linearity (ownership) arising from destructors.

References

1. Baker, H.G.: “Use-Once” Variables and Linear Objects - Storage Management, Reflection and Multi-Threading. SIGPLAN Notices **30**(1), 45–52 (1995). <https://doi.org/10.1145/199818.199860>
2. Baker, H.G.: Linear logic and permutation stacks - the Forth shall be first. SIGARCH Computer Architecture News **22**(1), 34–43 (1994). <https://doi.org/10.1145/181993.181999>
3. Baker, H.G.: Lively linear lisp: “Look ma, no garbage!” ACM Sigplan notices **27**(8), 89–98 (1992)
4. Benton, N., Kennedy, A.: Exceptional syntax. Journal of Functional Programming **11**, 395–410 (2001). <https://doi.org/10.1017/S0956796801004099>
5. Berdine, J., O’Hearn, P.W., Reddy, U.S., Thielecke, H.: Linearly used continuations. In: Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW’01), pp. 47–54 (2000)
6. Bernardy, J., Boespflug, M., Newton, R.R., Peyton Jones, S., Spiwack, A.: Linear Haskell: practical linearity in a higher-order polymorphic language. PACMPL **2**(POPL), 5:1–5:29 (2018). <https://doi.org/10.1145/3158093>
7. Blute, R., Cockett, J., Seely, R.: ! and ?-Storage as tensorial strength. Mathematical Structures in Computer Science **6**(4), 313–351 (1996)
8. Braüner, T.: A model of intuitionistic affine logic from stable domain theory. In: International Colloquium on Automata, Languages, and Programming, pp. 340–351 (1994)
9. Combette, G., Munch-Maccagnoni, G.: A resource modality for RAI (abstract). In: LOLA 2018: Workshop on Syntax and Semantics of Low-Level Languages (2018). <https://hal.inria.fr/hal-01806634>
10. Curien, P.-L., Fiore, M., Munch-Maccagnoni, G.: A Theory of Effects and Resources: Adjunction Models and Polarised Calculi. In: Proc. POPL (2016). <https://doi.org/10.1145/2837614.2837652>
11. Day, B.: On closed categories of functors. In: MacLane, S., Applegate, H., Barr, M., Day, B., Dubuc, E., Phreilambud, Pultr, A., Street, R., Tierney, M., Swierczkowski, S. (eds.) Reports of the Midwest Category Seminar IV, pp. 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg (1970)
12. Denis, X., Jourdan, J.-H., Marché, C.: Creusot: a Foundry for the Deductive Verification of Rust Programs. In: ICFEM 2022 - 23th International Conference on Formal Engineering Methods. LNCS, Springer, Heidelberg (2022). <https://inria.hal.science/hal-03737878>
13. Filinski, A.: Linear Continuations. In: Proc. POPL, pp. 27–38 (1992)
14. Filinski, A.: Representing Monads. In: Proc. POPL, pp. 446–457. ACM Press (1994)

15. Fluet, M., Morrisett, G., Ahmed, A.J.: Linear Regions Are All You Need. In: Sestoft, P. (ed.) Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27–28, 2006, Proceedings. LNCS, vol. 3924, pp. 7–21. Springer, Heidelberg (2006). https://doi.org/10.1007/11693024_2
16. Gan, E., Tov, J.A., Morrisett, G.: Type Classes for Lightweight Substructural Types. In: Alves, S., Cervesato, I. (eds.) Proceedings Third International Workshop on Linearity, LINEARITY 2014, Vienna, Austria, 13th July, 2014. EPTCS, pp. 34–48 (2014). <https://doi.org/10.4204/EPTCS.176.4>
17. Girard, J.-Y.: Linear Logic. Theoretical Computer Science **50**, 1–102 (1987)
18. Grossman, D., Morrisett, J.G., Jim, T., Hicks, M.W., Wang, Y., Cheney, J.: Region-Based Memory Management in Cyclone. In: Knoop, J., Hendren, L.J. (eds.) Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17–19, 2002, pp. 282–293. ACM (2002). <https://doi.org/10.1145/512529.512563>
19. Hasegawa, M.: Linear Exponential Comonads without Symmetry. In: Fourth International Workshop on Linearity (2016). <http://arxiv.org/abs/1701.04919>
20. Hasegawa, M.: Linearly Used Effects: Monadic and CPS Transformations into the Linear Lambda Calculus. In: Hu, Z., Rodríguez-Artalejo, M. (eds.) Functional and Logic Programming, 6th International Symposium, FLOPS 2002, Aizu, Japan, September 15–17, 2002, Proceedings. LNCS, vol. 2441, pp. 167–182. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45788-7_10
21. Hasegawa, M.: Semantics of linear continuation-passing in call-by-name. In: International Symposium on Functional and Logic Programming, pp. 229–243 (2004)
22. Hinnant, H.E., Dimov, P., Abrahams, D.: A Proposal to Add Move Semantics Support to the C++ Language, (2002). <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm>
23. Ho, S., Protzenko, J.: Aeneas: Rust Verification by Functional Translation. Proc. ACM Program. Lang. **6**(ICFP) (2022). <https://doi.org/10.1145/3547647>
24. Hofmann, M.: A Type System for Bounded Space and Functional In-Place Update. Nord. J. Comput. **7**(4), 258–289 (2000)
25. Jung, R., Jourdan, J., Krebbers, R., Dreyer, D.: RustBelt: securing the foundations of the rust programming language. PACMPL **2**(POPL), 66:1–66:34 (2018). <https://doi.org/10.1145/3158154>
26. Koenig, A., Stroustrup, B.: Exception Handling for C++. In: Proceedings of the C++ Conference. San Francisco, CA, USA, April 1990, pp. 149–176 (1990)
27. Lafont, Y.: The linear abstract machine. Theoretical computer science **59**(1–2), 157–180 (1988)
28. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2). Kluwer Academic Publishers, USA (2004)
29. Levy, P.B.: Call-by-Push-Value: A Subsuming Paradigm. In: Proc. TLCA '99, pp. 228–242 (1999)
30. Levy, P.B.: Contextual isomorphisms. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 400–414 (2017)
31. Maraist, J., Odersky, M., Turner, D.N., Wadler, P.: Call-by-Name, Call-by-Value, Call-by-Need, and the Linear Lambda Calculus. In: Proc. MFPS '95 (1994)
32. Melliès, P.-A.: “Categorical models of linear logic revisited”. working paper or preprint.
33. Melliès, P.-A.: “Categorical semantics of linear logic”. In: vol. 27. Panoramas et Synthèses. Société Mathématique de France, 2009. Chap. 1, pp. 15–215.
34. Melliès, P.-A.: “Parametric monads and enriched adjunctions”. Draft.

35. Møgelberg, R.E., Staton, S.: Linear usage of state. *Log. Methods Comput. Sci.* **10**(1) (2014). [https://doi.org/10.2168/LMCS-10\(1:17\)2014](https://doi.org/10.2168/LMCS-10(1:17)2014)
36. Moggi, E.: Computational lambda-calculus and monads. In: *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science (LICS 1989)*, pp. 14–23. IEEE Computer Society Press, Pacific Grove, CA, USA (1989)
37. Moggi, E.: Notions of computation and monads. *Information and Computation* **93**(1), 55–92 (1991). [https://doi.org/10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4). <https://www.sciencedirect.com/science/article/pii/0890540191900524>
38. Mostrous, D., Vasconcelos, V.T.: Affine Sessions. *Logical Methods in Computer Science*, Volume 14, Issue 4 (November 15, 2018) *lmcs*:4973 (2018). [https://doi.org/10.23638/LMCS-14\(4:14\)2018](https://doi.org/10.23638/LMCS-14(4:14)2018). arXiv: 1809.02781v2 [cs.LO]
39. Munch-Maccagnoni, G.: “Resource Polymorphism”.
40. Orchard, D., Liepelt, V.-B., Eades III, H.: Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* **3**(ICFP) (2019). <https://doi.org/10.1145/3341714>
41. Osvald, L., Essertel, G., Wu, X., Alayón, L.I.G., Rompf, T.: Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. SPLASH '16*, pp. 234–251. ACM (2016). <https://doi.org/10.1145/2983990.2984009>
42. Pfenning, F., Simmons, R.J.: Substructural Operational Semantics as Ordered Logic Programming. In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pp. 101–110. IEEE Computer Society (2009). <https://doi.org/10.1109/LICS.2009.8>
43. Plotkin, G., Power, J.: “Notions of Computation Determine Monads”. In: *Foundations of Software Science and Computation Structures*. Springer Berlin Heidelberg, 2002, pp. 342–356. ISBN: 9783540459316. https://doi.org/10.1007/3-540-45931-6_24.
44. Polakow, J.: Ordered Linear Logic and Applications. PhD thesis, Carnegie Mellon University (2001).
45. Polakow, J., Yi, K.: Proving Syntactic Properties of Exceptions in an Ordered Logical Framework. In: Kuchen, H., Ueda, K. (eds.) *Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 7-9, 2001, Proceedings. LNCS*, vol. 2024, pp. 61–77. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44716-4_4
46. Power, A.J., Robinson, E.: Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science* **5**(7), 453–468 (1997)
47. Power, J.: Premonoidal categories as categories with algebraic structure. *Theoretical Computer Science* **278**(1), 303–321 (2002). [https://doi.org/10.1016/S0304-3975\(00\)00340-6](https://doi.org/10.1016/S0304-3975(00)00340-6). <https://www.sciencedirect.com/science/article/pii/S0304397500003406>
48. Selinger, P., Valiron, B.: A linear-non-linear model for a computational call-by-value lambda calculus. *Lecture Notes in Computer Science* **4962**, 81–96 (2008)
49. Stroustrup, B.: A History of C++: 1979–1991. In: *The Second ACM SIGPLAN Conference on History of Programming Languages. HOPL-II*, pp. 271–297. Association for Computing Machinery, Cambridge, Massachusetts, USA (1993). <https://doi.org/10.1145/154766.155375>
50. Swamy, N., Hicks, M., Morrisett, G., Grossman, D., Jim, T.: Safe manual memory management in Cyclone. *Science of Computer Programming* **62**(2), 122–144 (2006)
51. Tofte, M., Talpin, J.-P.: Region-based memory management. *Information and computation* **132**(2), 109–176 (1997)
52. Tov, J.A.: Practical Programming with Substructural Types. PhD thesis, Northeastern University (2012).
53. Tov, J.A., Pucella, R.: A Theory of Substructural Types and Control. In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages*

- and Applications. OOPSLA '11, pp. 625–642. ACM, Portland, Oregon, USA (2011). <https://doi.org/10.1145/2048066.2048115>. <http://doi.acm.org/10.1145/2048066.2048115>
54. Tov, J.A., Pucella, R.: Practical affine types. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011, pp. 447–458. ACM (2011). <https://doi.org/10.1145/1926385.1926436>
55. Walker, D.: Substructural type systems. In: Advanced Topics in Types and Programming Languages. Ed. by B.C. Pierce, pp. 3–44. The MIT Press (2005)

A Context: resource management in programming

Programmers must sometimes deal with *resources*: values denoting some memory allocations, files, locks, etc.—in general reifying some knowledge about the state of the world—that must be disposed of at specific points in the program to ensure its proper functioning. For instance, some task might operate on a file descriptor, obtained and released through operating system calls, as illustrated with the following pseudo-code:

```
let f = open_file("record.txt");
append(f, "starting tasks");
run_tasks(f);
close_file(f);
```

Listing 1.1. Opening and closing a file

It is generally a good idea to close files in a timely manner: for instance, depending on the operating system and its command, the number of files that the program is allowed to keep opened concurrently can be fairly low. Another common phenomenon affecting resources is that their acquisition can fail, which actually compounds the problem of correctly disposing of resources: it must be carefully done in case of failure of later resource acquisitions. This is where C programmers resort to using `goto`—one of the only situations where it is not only allowed but idiomatic; in pseudo-code:

```
let f = open_file("record.txt");
if !f return;
append(f, "starting tasks");
let g = open_file("output.txt");
if !g goto err;
run_tasks(f, g);
close_file(g);
err:
close_file(f);
```

Listing 1.2. Handling errors with `goto`

Here, `f` is closed whether opening `g` succeeds or not.

Managing resources, especially explicitly as in C, comes with specific concerns:

1. disposing of the resource in a timely fashion in all the code paths (avoiding leaks, as we have seen),

2. keeping track of who is responsible for disposing of the resource (to avoid for instance that the resource is disposed of twice),
3. making sure that a resource is never used after its disposal (*use-after-free*).

In simple situations like the previous one, and in languages that support it, the resources can be managed automatically with higher-order scoped combinators derived from LISP's `unwind-protect`:

```
with_file("record.txt", λ f.
  append(f, "starting tasks");
  with_file("output.txt", λ g.
    run_tasks(f, g)
  );
);
```

Listing 1.3. Handling errors with higher-order scoped combinators

In words, the combinator `with_file` opens the file and passes the file descriptor to its argument. It closes the file upon normal or exceptional return, and returns the same value that is returned by the functional argument.

Now, this approach with scoped combinators comes with several limitations. Firstly, they force resources to be disposed of in a strict LIFO order. They do not model situations where the duty of clean-up is transferred to the caller or to a callee, nor where the resource is stored in a data structure. This limits expressiveness but also compositionality (the ability to reorganise the program in smaller and more general parts). In effect, scoped combinators solve (1) and (2) above albeit in a heavy-handed way.

The second limitation has to do with the transient nature of the value they pass to their functional argument: this argument is indeed meaningful for the duration of the scope only, since using the value after the scope ends (by sneaking it inside the return value, or by storing it inside some mutable data structure) is an attempt to use the resource after its disposal (3). Yet, keeping track of the value can be far from obvious, especially when programming using other (*e.g.* monadic) combinators.

This paper focuses on methods for lifting the limitation using a notion of *ownership* keeping track of who is responsible for clean-up, generally understood to be related to linearity. In fact, we elaborate a formal link between an advanced notion of ownership present in the C++ and Rust programming languages, and linearity. We do not try to model the second limitation, which involve other tools, and is already the subject of much study, for instance regions [51, 15], which inspired Rust lifetimes via Cyclone [18, 50], or second-class values [41] (See section 6). We focus on the first problem by studying a notion of resources that have no use apart from acquisition and release: hence they cannot be used after being released indeed.

A.1 C++ destructors

Independently from linear types, modern systems programming languages such as C++ and Rust have since the 1980's evolved automatic resource-management features that not only combine well with error management and control effects, but in which error handling is an essential part—indeed resource management requires to deal with the

correct disposal of resources, but also the handling of possible failure of their allocation. Furthermore, links with linear logic were suggested as early as [2], but remained elusive.

Each type A comes equipped with a *destructor*, a clean-up action that removes a value of A . It is called when such variable goes out of scope, because of an explicit return or an exception. Destructors provide a notion of ownership, tying resources to variables lifetime. Local variables hence yields a LIFO ordering for handling resources. The previous example would be written like this in C++:

```
File f{"record.txt"};
append(f, "starting tasks");
File g{"output.txt"};
run_tasks(f, g);
// g then f are then closed by their destructors
```

Listing 1.4. Handling errors with destructors

Sometimes, the control flow of resources becomes more complex. For example, in the following example, we transfer the file ownership to `run_task`, in the case where tasks have been initialised. It then becomes difficult to track the places responsible to call `close_file`, leading to potential leaks or errors. Also, this use case does not fit in the pattern of higher-order combinators, which would always close the file at the end of the scope.

```
let f = open_file("log.txt");
if !f return;
if init_tasks() == ok:
    append(f, "starting tasks");
    run_tasks(f);
else:
    append(f, "failure to init tasks");
    close_file(f);
```

Listing 1.5. Conditional transfer of manual ownership

After empirical investigations with `auto_ptr`, the C++11 standard added to the language a move operation to transfer ownership of values, as shown below:

```
File f{"log.txt"};
if init_tasks() == ok {
    append(f, "starting tasks");
    run_tasks(std::move(f));
} else {
    append(f, "failure to init tasks");
}
```

Listing 1.6. Pseudo-code using a file to record operations

C++ datatypes combine destructors of their members to treat resources as first-class values in its abstractions:

- Pairs, tuples and records call destructors of their components in sequence.

- `std::variants` (sum types) call the destructor of the active component.
- Collections call the destructors of each to their elements.
- Closures get a dedicated type which destructs captured values.

B Elementary properties

B.1 Proof of swap and definition of swap inverse

We recall the expressions swap_W^A :

$$\text{swap}_I^A \stackrel{\text{def}}{=} \lambda p. \delta(p, (a, i). \delta(i, (). (\star, a)))$$

$$\begin{aligned} \text{swap}_{W_1 \oplus W_2}^A &\stackrel{\text{def}}{=} \lambda p. \delta(p, (a, w). \delta(w, \\ &\quad w_1. \text{let } p_1 = \text{swap}_{W_1}^A(a, w_1) \text{ in } \delta(p_1, (w_1, a). (\iota_1 w_1, a)), \\ &\quad w_2. \text{let } p_2 = \text{swap}_{W_2}^A(a, w_2) \text{ in } \delta(p_2, (w_2, a). (\iota_2 w_2, a)))) \end{aligned}$$

$$\begin{aligned} \text{swap}_{W_1 \otimes W_2}^A &\stackrel{\text{def}}{=} \lambda p. \delta(p, (a, w). \delta(w, (w_1, w_2). \\ &\quad \text{let } p_1 = \text{swap}_{W_2}^{A \otimes W_1}((a, w_1), w_2) \text{ in } \delta(p_1, (w_2, p). \delta(p, (a, w_1). \\ &\quad \text{let } p_2 = \text{swap}_{W_1}^{W_2 \otimes A}((w_2, a), w_1) \text{ in } \delta(p_2, (w_1, p). \delta(p, (w_2, a). ((w_1, w_2), a))))))) \end{aligned}$$

We define similarly the expressions $\text{swap}_W'^A : A \otimes W \multimap W \otimes A$:

$$\text{swap}_I'^A \stackrel{\text{def}}{=} \lambda p. \delta(p, (i, a). \delta(i, (). (a, \star)))$$

$$\begin{aligned} \text{swap}_{W_1 \oplus W_2}'^A &\stackrel{\text{def}}{=} \lambda p. \delta(p, (w, a). \delta(w, \\ &\quad w_1. \text{let } p_1 = \text{swap}_{W_1}'^A(w_1, a) \text{ in } \delta(p_1, (a, w_1). (a, \iota_1 w_1)), \\ &\quad w_2. \text{let } p_2 = \text{swap}_{W_2}'^A(w_2, a) \text{ in } \delta(p_2, (a, w_2). (a, \iota_2 w_2)))) \end{aligned}$$

$$\begin{aligned} \text{swap}_{W_1 \otimes W_2}'^A &\stackrel{\text{def}}{=} \lambda p. \delta(p, (w, a). \delta(w, (w_1, w_2). \\ &\quad \text{let } p_1 = \text{swap}_{W_2}'^{A \otimes W_1}(w_1, (w_2, a)) \text{ in } \delta(p_1, (p, w_1). \delta(p, (w_2, a). \\ &\quad \text{let } p_2 = \text{swap}_{W_1}'^{W_2 \otimes A}(w_2, (a, w_1)) \text{ in } \delta(p_2, (p, w_2). \delta(p, (a, w_1). (a, (w_1, w_2))))))) \end{aligned}$$

Those terms are well-typed: variables are used exactly once and in order. Intro and elimination rules for positive types do not have ordering constraints. Those from function application and let bindings are respected: swap arguments use right-most variables. Informally, those terms implement the following operations on stacks, with permutations restricted to the two right-most types:

- $\text{swap}_I^A \stackrel{\text{def}}{=} AI \rightarrow A \rightarrow IA.$
- $\text{swap}_{W_1 \oplus W_2}^A \stackrel{\text{def}}{=} A(W_1 \oplus W_2) \rightarrow AW_i \rightarrow W_i A \rightarrow (W_1 \oplus W_2)A.$
- $\text{swap}_{W_1 W_2}^A \stackrel{\text{def}}{=} A(W_1 W_2) \rightarrow (AW_1)W_2 \rightarrow W_2(AW_1) \rightarrow (W_2 A)W_1 \rightarrow W_1(W_2 A) \rightarrow (W_1 W_2)A.$
- $\text{swap}_A^I \stackrel{\text{def}}{=} IA \rightarrow A \rightarrow AI.$
- $\text{swap}_{W_1 \oplus W_2}^A \stackrel{\text{def}}{=} (W_1 \oplus W_2)A \rightarrow W_i A \rightarrow AW_i \rightarrow A(W_1 \oplus W_2).$
- $\text{swap}_A^{W_1 W_2} \stackrel{\text{def}}{=} (W_1 W_2)A \rightarrow W_1(W_2 A) \rightarrow (W_2 A)W_1 \rightarrow W_2(AW_1) \rightarrow (AW_1)W_2 \rightarrow A(W_1 W_2).$

B.2 Proof of determinism (theorem 1)

This is shown by case analysis, first on the command polarity:

For negative commands, all expressions are mutually exclusive except the two “new” cases, for which the lists are mutually exclusive (empty or not).

For positive commands, all expressions are mutually exclusive once we analyse v in $\langle v \mid (x^+.t) \cdot s \mid l \rangle^+ \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle^+$: either the value is a variable or positive, in which case no other rules apply, or the value is negative and may reduce only in negative commands.

B.3 Proof of subject reduction (theorem 3)

Show that $\forall c1 : A, \forall c2, c1 \rightsquigarrow c2 \Rightarrow c2 : A$ by case analysis on the reduction step:

- $\langle (\text{let } x^+ = t \text{ in } u)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t \mid (x^+.u)^\epsilon \cdot s \mid l \rangle^+$: by the hypothesis $\langle (\text{let } x^+ = t \text{ in } u)^\epsilon \mid s \mid l \rangle^\epsilon : A$, we get $\vdash_p t : B_+, x : B \vdash_p u : C_\epsilon$ and $s : C \vdash_p A$. Hence, we have $(x^+.u)^\epsilon \cdot s : B \vdash_p A$ and so $\langle t \mid (x^+.u)^\epsilon \cdot s \mid l \rangle^+ : A$.
- $\langle (\text{let } x^- = v \text{ in } t)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle^\epsilon$: by hypothesis, we get $\vdash_p v : B_-, x : B \vdash_p t : C_\epsilon$ and $s : C \vdash_p A$. By SL, we have $\vdash_p t[v/x] : C$ and so $\langle t[v/x] \mid s \mid l \rangle^\epsilon : A$.
- $\langle (vw)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle v \mid w^\epsilon \cdot s \mid l \rangle^-$: by hypothesis, we get $\vdash_p v : C_\epsilon \multimap B, \vdash_p w : B$ and $s : C \vdash_p A$. Hence, we have $w^\epsilon \cdot s : C \multimap B \vdash_p A$ and so $\langle v \mid w^\epsilon \cdot s \mid l \rangle^- : A$.
- $\langle (\pi_i v)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle v \mid \pi_i^\epsilon \cdot s \mid l \rangle^-$: by hypothesis, we get $\vdash_p v : B_1 \& B_2$ and $s : B_{i\epsilon} \vdash_p A$. Hence, we have $\pi_i^\epsilon \cdot s : B_1 \& B_2 \vdash_p A$ and so $\langle v \mid \pi_i^\epsilon \cdot s \mid l \rangle^- : A$.
- $\langle v \mid (x^+.t)^\epsilon \cdot s \mid l \rangle^+ \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle^\epsilon$: by hypothesis, we get $\vdash_p v : B_+, x : B \vdash_p t : C$ and $s : C \vdash_p A$. By SL, we have $\vdash_p t[v/x] : C$ and so $\langle t[v/x] \mid s \mid l \rangle^\epsilon : A$.
- $\langle \lambda x. t \mid v^\epsilon \cdot s \mid l \rangle^- \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle^\epsilon$: by hypothesis, we get $x : B \vdash_p t : C_\epsilon, \vdash_p v : B$ and $s : C \vdash_p A$. By SL, we have $\vdash_p t[v/x] : C$ and so $\langle t[v/x] \mid s \mid l \rangle^\epsilon : A$.
- $\langle \langle t_1, t_2 \rangle \mid \pi_i^\epsilon \cdot s \mid l \rangle^- \rightsquigarrow \langle t_i \mid s \mid l \rangle^\epsilon$: by hypothesis, we get $\vdash_p t_1 : B_1, \vdash_p t_2 : B_2$ and $s : B_{i\epsilon} \vdash_p A$. Hence, we have $\langle t_i \mid s \mid l \rangle^\epsilon : A$.
- $\langle \delta((v, w), (x, y).t)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t[v/x, w/y] \mid s \mid l \rangle^\epsilon$: by hypothesis, we get $\vdash_p v : B, \vdash_p w : C, x : B, y : C \vdash_p t : D_\epsilon$ and $s : D \vdash_p A$. By SL, we have $\vdash_p (t[v/x])[w/y] : D$. Because x nor y do not occur free in v nor w , we have $(t[v/x])[w/y] = t[v/x, w/y]$ and so $\langle t[v/x, w/y] \mid s \mid l \rangle^\epsilon : A$.
- $\langle \delta((), ().t)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t \mid s \mid l \rangle^\epsilon$: by hypothesis, we get $\vdash_p t : B_\epsilon$ and $s : B \vdash_p A$, hence we have $\langle t \mid s \mid l \rangle^\epsilon : A$.

- $\langle \delta(t_i v, x_1.t_1, x_2.t_2)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t_i[v/x_i] \mid s \mid l \rangle^\epsilon$: by hypothesis, we get $\vdash_p v : B_i, x_1 : B_1 \vdash_p t_1 : C_\epsilon, x_2 : B_2 \vdash_p t_2 : C$ and $s : C \vdash_p A$. By SL, we have $\vdash_p t_i[v/x_i] : C$ and so $\langle t_i[v/x_i] \mid s \mid l \rangle^\epsilon : A$.
- $\langle \text{new} \mid () \cdot s \mid r_n :: l \rangle^- \rightsquigarrow \langle t_1 r_n \mid s \mid l \rangle^+$: by hypothesis, we have $s : R \oplus 1 \vdash_p A$. We always have $\vdash_p t_1 r_n : R \oplus 1$, and so $\langle t_1 r_n \mid s \mid l \rangle^+ : A$.
- $\langle \text{new} \mid () \cdot s \mid \text{Nil} \rangle^- \rightsquigarrow \langle t_2 () \mid s \mid \text{Nil} \rangle^+$: by hypothesis, we have $s : R \oplus 1 \vdash_p A$. We always have $\vdash_p t_2 () : R \oplus 1$, and so $\langle t_1 r_n \mid s \mid l \rangle^+ : A$.
- $\langle \text{delete} \mid r_n \cdot s \mid l \rangle^- \rightsquigarrow \langle () \mid s \mid r_n :: l \rangle^+$: by hypothesis, we have $s : 1 \vdash A$. We always have $\vdash_p () : 1$ and so $\langle () \mid s \mid r_n :: l \rangle^+ :$.

The same proof goes through if we consider $\vdash_p$ without structural rules.

B.4 Proof of progress (theorem 4)

Show that any well-typed command $\langle t \mid s \mid l \rangle^\epsilon : A$ reduces iff it is not of the shape $\langle v_t \mid \star \mid l \rangle^\epsilon$ by case analysis on the judgement $\vdash_p t : B_\epsilon$:

If t is a final value, we do a case analysis on the judgement $s : B \vdash_p A$: if $s = \star$, the context is of the shape $\langle v_t \mid \star \mid l \rangle^\epsilon$ and matches no reduction steps. Otherwise, the stack shape is determined by the type of t :

- If $\varpi(B) = +$, then $s = (x^+.u)^\epsilon \cdot s'$ and $\langle v_t \mid (x^+.u)^\epsilon \cdot s' \mid l \rangle^+ \rightsquigarrow \langle u[v_t/x] \mid s' \mid l \rangle^\epsilon$.
- If $t = \langle t_1, t_2 \rangle$, then $s = \pi_i^\epsilon \cdot s'$ and $\langle \langle t_1, t_2 \rangle \mid \pi_i^\epsilon \cdot s' \mid l \rangle^- \rightsquigarrow \langle t_i \mid s' \mid l \rangle^\epsilon$.
- If $t = \lambda x. t'$, then $s = v^\epsilon \cdot s'$ and $\langle \lambda x. t' \mid v^\epsilon \cdot s' \mid l \rangle^- \rightsquigarrow \langle t'[v/x] \mid s' \mid l \rangle^\epsilon$.
- If $t = \text{new}$ and $l = r_n :: l'$, then $s = () \cdot s'$ and $\langle \text{new} \mid () \cdot s' \mid r_n :: l' \rangle^- \rightsquigarrow \langle t_1 r_n \mid s' \mid l' \rangle^+$.
- If $t = \text{new}$ and $l = \star$, then $s = () \cdot s'$ and $\langle \text{new} \mid () \cdot s' \mid \star \rangle^- \rightsquigarrow \langle t_2 () \mid s' \mid \star \rangle^+$.
- If $t = \text{delete}$ so $s = r_n \cdot s'$ and $\langle \text{delete} \mid r_n \cdot s' \mid l \rangle^- \rightsquigarrow \langle () \mid s' \mid r_n :: l \rangle^+$.

Otherwise:

- If $t = \delta((v, w), (x, y).t')^\epsilon$, then $\langle \delta((v, w), (x, y).t')^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t'[v/x, w/y] \mid s \mid l \rangle^\epsilon$.
- If $t = \delta((), ().t')^\epsilon$, then $\langle \delta((), ().t')^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t' \mid s \mid l \rangle^\epsilon$.
- If $t = \delta(t_i v, x_1.t_1, x_2.t_2)^\epsilon$, then $\langle \delta(t_i v, x_1.t_1, x_2.t_2)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t_i[x_i/v] \mid s \mid l \rangle^\epsilon$.
- If $t = (\text{let } x^- = v \text{ in } t')^\epsilon$, then $\langle (\text{let } x^- = v \text{ in } t')^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t'[v/x] \mid s \mid l \rangle^\epsilon$.
- If $t = (\text{let } x^+ = t' \text{ in } u)^\epsilon$, then $\langle (\text{let } x^+ = t' \text{ in } u)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle t' \mid (x^+.u)^\epsilon \cdot s \mid l \rangle^+$.
- If $t = (vw)^\epsilon$, then $\langle (vw)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle v \mid w^\epsilon \cdot s \mid l \rangle^-$.
- If $t = (\pi_i v)^\epsilon$, then $\langle (\pi_i v)^\epsilon \mid s \mid l \rangle^\epsilon \rightsquigarrow \langle v \mid \pi_i^\epsilon \cdot s \mid l \rangle^-$.

The same proof goes through if we consider $\vdash_p$ without structural rules.

C Resource-safety properties

C.1 Proof that $\vdash$ implies resource-free $\vdash_o$ (lemma 6)

Proof. By induction on the derivation of t , we apply the corresponding rule in $\vdash_o$: contexts of premises are always well-formed due to the absence of resources. Rules to shift between the contexts $\Gamma, x; []; \Delta$ and $\Gamma; [], x, \Delta$ are used as needed, in particular at the end to obtain $\Gamma; ; \vdash t$.

C.2 Proof of left and right substitution lemmas for ordered expressions

(lemma 7)

Show that (Left SL) if $; L_t; \vdash_o t$ and $\Gamma; L_u; \Delta \vdash_o u$, then $\Gamma; L_t \# L_u; \Delta \vdash_o u[t/x]$.

Moreover (Right SL), if $; L_t; \vdash_o t$ and $\Gamma; L_u; x, \Delta \vdash_o u$, then $\Gamma; L_u \# L_t; \Delta \vdash_o u[t/x]$.

We first prove by induction that if a variable x does not occur in an expression u , then $u[t/x] = u$. Then, each SL is proven by induction on u :

- $()$, **new**, **delete**, r_n : both SL by absurdity from the induction hypothesis (IH): no variables in the context.
- $x \vdash_o x$: for both SL, we have $; L_t; \vdash_o t$ and $x[t/x] = t$ so we have $; L_t; \vdash_o x[t/x]$.
- $\iota_i v$: for left SL, by IH we have $\Gamma; L_t \# L_u; \Delta \vdash_o v[t/x]$. Because $(\iota_i v)[t/x] = \iota_i(v[t/x])$, we have $\Gamma; L_t \# L_u; \Delta \vdash_o (\iota_i v)[t/x]$. The right SL is done similarly, swapping L_t and L_u : by IH we have $\Gamma; L_u \# L_t; \Delta \vdash_o v[t/x]$. Because $(\iota_i v)[t/x] = \iota_i(v[t/x])$, we have $\Gamma; L_u \# L_t; \Delta \vdash_o (\iota_i v)[t/x]$.
- $\pi_i v$: similarly, because $(\pi_i v)[t/x] = \pi_i(v[t/x])$ both SL hold by IH.
- $\langle u_1, u_2 \rangle$: because $\langle u_1, u_2 \rangle[t/x] = \langle u_1[t/x], u_2[t/x] \rangle$, both SL hold by IH.
- $\lambda y. u_1$: because $\lambda y. (u_1[t/x]) = (\lambda y. u_1)[t/x]$, both SL hold by IH.
- (v, w) : for both SL, given $\Theta \vdash_o v$ and $\Theta' \vdash_o w$, by case analysis on the three cases in which $\Theta @ \Theta'$ is well-formed. In the left SL case:
 - $\Theta = \Gamma_v; ;$ and $\Theta' = \Gamma_w^1; x; L_u; \Gamma_w^2$: x does not occur in v so $(v, w)[t/x] = (v, w[t/x])$ and by IH on w we have $\Gamma_w^1; x; L_u; \Gamma_w^2 \vdash_o w[t/x]$, hence we have $\Gamma_v, \Gamma_w^1; L_t \# L_u; \Gamma_w^2 \vdash_o (v, w)[t/x]$.
 - $\Theta = \Gamma_v^1; x; L_u; \Gamma_v^2$ and $\Theta' = \Gamma_w; ;$: $(v, w)[t/x] = (v[t/x], w)$ so the SL holds by IH on $\Theta \vdash_o v$.
 - $\Theta = \Gamma_v; x; L_u^1$ and $\Theta' = L_u^2; \Gamma_w$: $(v, w)[t/x] = (v[t/x], w)$ so the SL holds by IH on v .

The right SL is done similarly, swapping x and L_u :

- $\Theta = ;$; Γ_v and $\Theta' = \Gamma_w^1; L_u; x; \Gamma_w^2$: $(v, w)[t/x] = (v, w[t/x])$ so the SL holds by IH on w .
- $\Theta = \Gamma_v^1; L_u; x; \Gamma_v^2$ and $\Theta' = \Gamma_w$: $(v, w)[t/x] = (v[t/x], w)$ so the SL holds by IH on v .
- $\Theta = \Gamma_v; L_u^1$ and $\Theta' = L_u^2; x; \Gamma_w$: $(v, w)[t/x] = (v, w[t/x])$ so the SL holds by IH on w .
- let $y = u_1$ in u_2 : similarly, for both SL we proceed by case analysis on $\Theta @ \Theta'$. For the left SL:
 - $\Theta = ;$; Γ_{u_2}, y and $\Theta' = \Gamma_{u_1}^1; x; ; L_u; \Gamma_{u_1}^2$: $(\text{let } y = u_1 \text{ in } u_2)[t/x] = (\text{let } y = u_1 \text{ in } u_2[t/x])$ so the SL holds by IH on u_2 .
 - $\Theta = \Gamma_{u_2}^1; x; L_u; \Gamma_{u_2}^2, y$ and $\Theta' = ;$; Γ_w : $(\text{let } y = u_1 \text{ in } u_2)[t/x] = (\text{let } y = u_1[t/x] \text{ in } u_2)$ so the SL holds by IH on u_1 .
 - $\Theta = \Gamma_{u_2}; L_u^1; y$ and $\Theta' = L_u^2; x; \Gamma_{u_1}$: the concatenation is not well-formed due to y so the SL holds by absurdity.
- The right SL is done similarly by swapping x and L_u .
- $(v, y. u_1, z. u_2)$: For both SL we proceed by case analysis on $\Theta @ \Theta' @ \Theta''$, here for left SL:
 - $\Theta = \Gamma_u^1; x; L_u; \Gamma_u^2, \Theta' = ;$; $\Gamma_v, \Theta'' = ;$; Γ_u^3 : $(v, y. u_1, z. u_2)[t/x] = (v, y. u_1[t/x], z. u_2[t/x])$ so the SL holds by IH on $\Gamma_u^1; x; L_u; \Gamma_u^2, y, \Gamma_u^3 \vdash_o u_1$ and $\Gamma_u^1; x; L_u; \Gamma_u^2, y, \Gamma_u^3 \vdash_o u_2$.

- $\Theta = \Gamma_u^1, x; L_u^1, \Theta' =; L_u^2; \Gamma_v, \Theta'' =; \Gamma_u^2$: the SL holds by IH on u_1 and u_2 .
- $\Theta =; \Gamma_u^1, \Theta' = \Gamma_v^1, x; L_u; \Gamma_v^2, \Theta'' =; \Gamma_u^2$: the SL holds by IH on v .
- $\Theta =; \Gamma_u^1, \Theta' = \Gamma_v^1, x; L_u^1; \Theta'' =; L_u^2; \Gamma_u^2$: the SL holds by IH on v .
- $\Theta =; \Gamma_u^1, \Theta' =; \Gamma_v, \Theta'' = \Gamma_u^2, x; L_u; \Gamma_u^3$: the SL holds by IH on u_1 and u_2 .
- The other cases yield ill-formed concatenated context.

The right SL is done similarly by swapping x and L_u .

- $\delta(v, (y, z).u_1)$: similarly, by case analysis on $\Theta @ \Theta' @ \Theta''$.
- $\delta(v, ().t)$: by case analysis on $\Theta @ \Theta'$.
- vw : by case analysis on $\Theta @ \Theta'$.

C.3 Proof of ordered predicate invariant by reduction (theorem 8)

Show that $\forall c1, \forall c2, \forall L, L \vdash_o^C c1 \wedge c1 \rightsquigarrow c2 \Rightarrow L \vdash_o^C c2$ by case analysis on reduction steps:

- $\langle \text{let } x^+ = t \text{ in } u \mid s \mid l \rangle \rightsquigarrow \langle t \mid (x^+.u) \cdot s \mid l \rangle$: by definition of $L \vdash_o^C c1$, we get $; L_t; \vdash_o t$; $; L_u; x \vdash_o u$ and $L_s \vdash_o^S s$ with $L = L_s \uplus L_u \uplus L_t \uplus l$, hence we have $; L_s \uplus L_u; \vdash_o (x^+.u) \cdot s$ and so $L_s \uplus L_u \uplus L_t \uplus l = L \vdash_o^C c2$.
- $\langle \text{let } x^- = t \text{ in } u \mid s \mid l \rangle \rightsquigarrow \langle u[t/x] \mid s \mid l \rangle$: by definition, we get $; L_t; \vdash_o t$; $; L_u; x \vdash_o u$ and $L_s \vdash_o^S s$ hence by right SL we have $; L_u \uplus L_t; \vdash_o u[t/x]$ and so $L \vdash_o^C c2$.
- $\langle vw \mid s \mid l \rangle \rightsquigarrow \langle v \mid w \cdot s \mid l \rangle$: by definition, we get $; L_w; \vdash_o w$; $; L_v; \vdash_o v$ and $L_s \vdash_o^S s$, hence we have $L_s \uplus L_w \vdash_o^S w \cdot s$ and so $L \vdash_o^C c2$.
- $\langle \pi_i t \mid s \mid l \rangle \rightsquigarrow \langle t \mid \pi_i \cdot s \mid l \rangle$: by definition, we get $; L_t; \vdash_o t$ and $L_s \vdash_o^S s$, hence we have $L \vdash_o^C c2$.
- $\langle v \mid (x.t) \cdot s \mid l \rangle \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle$: by definition, we get $; L_v; \vdash_o v$; $; L_t; x \vdash_o t$ and $L_s \vdash_o^S s$, hence by right SL we have $; L_t \uplus L_v; \vdash_o t[v/x]$ and so $L \vdash_o^C c2$.
- $\langle \lambda x. t \mid v \cdot s \mid l \rangle \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle$: by definition, we get $L_v \vdash_o v$, $x, L_t \vdash_o t$ and $L_s \vdash_o^S s$, hence by left SL we have $L_v, L_t \vdash_o t[v/x]$ and so $L \vdash_o^C c2$.
- $\langle \langle t_1, t_2 \rangle \mid \pi_i \cdot s \mid l \rangle \rightsquigarrow \langle t_i \mid s \mid l \rangle$: by definition, we get $L_t \vdash_o t_1$, $L_t \vdash_o t_2$ and $L_s \vdash_o^S s$, hence we have $L \vdash_o^C c2$.
- $\langle \delta((v, w), (x, y).t) \mid s \mid l \rangle \rightsquigarrow \langle t[v/x, w/y] \mid s \mid l \rangle$: by definition and case analysis on composition of contexts, we get $; L_v; \vdash_o v$; $; L_w; \vdash_o w$, $\Theta \vdash_o t$ and $L_s \vdash_o^S s$ with $\Theta =$ either $; L_t; x, y \text{ or } x, y; L_t$. In the first case, by right SL twice we have $; L_t \uplus L_v \uplus L_w; \vdash_o t[v/x, w/y]$ and so $L \vdash_o^C c2$. Similarly, in the second case by left SL twice we have $; L_v \uplus L_w \uplus L_t; \vdash_o t[v/x, w/y]$ and so $L \vdash_o^C c2$.
- $\langle \delta((), ().t) \mid s \mid l \rangle \rightsquigarrow \langle t \mid s \mid l \rangle$: by definition, we get $; []; \vdash_o ()$; $; L_t; \vdash_o t$ and $L_s \vdash_o^S s$, hence we have $L \vdash_o^C c2$.
- $\langle \delta(t_i v, x_1.t_1, x_2.t_2) \mid s \mid l \rangle \rightsquigarrow \langle t_i[x_i/v] \mid s \mid l \rangle$: by definition and case analysis, we get $; L_v; \vdash_o v$, $\Theta_i \vdash_o t_i$ and $L_s \vdash_o^S s$ with both $\Theta_i =$ either $; L_t; x_i \text{ or } x_i; L_t$. In the first case, by right SL we have $; L_t \uplus L_v; \vdash_o t_i[x_i/v]$ and so $L \vdash_o^C c2$. Similarly, in the second case by left SL we have $; L_v \uplus L_t; \vdash_o t_i[x_i/v]$ and so $L \vdash_o^C c2$.
- $\langle \text{new} \mid () \cdot s \mid r_n :: l \rangle \rightsquigarrow \langle t_1 r_n \mid s \mid l \rangle$: by definition, we get $; []; \vdash_o ()$; $; [r_n]; \vdash_o r_n$ and $L_s \vdash_o^S s$, hence we have $L_s \uplus r_n :: l = L \vdash_o^C c2$.
- $\langle \text{new} \mid () \cdot s \mid \text{Nil} \rangle \rightsquigarrow \langle t_2 () \mid s \mid \text{Nil} \rangle$: by definition, we get $L_s = L \vdash_o^C c2$.
- $\langle \text{delete} \mid r_n \cdot s \mid l \rangle \rightsquigarrow \langle () \mid s \mid r_n :: l \rangle$: by definition, we get $; [r_n]; \vdash_o r_n$; $; []; \vdash_o ()$ and $L_s \vdash_o^S s$, hence we have $L_s \uplus r_n :: l = L \vdash_o^C c2$.

$\overline{[]; x \vdash_l x}$	$\overline{[]; \vdash_l ()}$	$\overline{[]; \vdash_l \text{new}}$	$\overline{[]; \vdash_l \text{delete}}$	$\overline{[r_n]; \vdash_l r_n}$
$\frac{\Theta \vdash_l v}{\Theta \vdash_l \iota_i v}$	$\frac{\Theta \vdash_l v}{\Theta \vdash_l \pi_i v}$	$\frac{\Theta \vdash_l t \quad \Theta \vdash_l u}{\Theta \vdash_l \langle t, u \rangle}$	$\frac{\Theta, x \vdash_l t}{\Theta \vdash_l \lambda x. t}$	$\frac{\Theta \vdash_l v \quad \Theta' \vdash_l w}{\Theta @ \Theta' \vdash_l (v, w)}$
$\frac{\Theta, x \vdash_l u \quad \Theta' \vdash_l t}{\Theta @ \Theta' \vdash_l \text{let } x = t \text{ in } u}$	$\frac{\Theta, x, y \vdash_l t \quad \Theta' \vdash_l v}{\Theta @ \Theta' \vdash_l \delta(v, (x, y).t)}$	$\frac{\Theta \vdash_l t \quad \Theta' \vdash_l v}{\Theta @ \Theta' \vdash_l \delta(v, ().t)}$		
$\frac{\Theta, x \vdash_l t \quad \Theta, y \vdash_l u \quad \Theta' \vdash_l v}{\Theta @ \Theta' \vdash_l \delta(v, x.t, y.u)}$			$\frac{\Theta \vdash_l w \quad \Theta' \vdash_l v}{\Theta @ \Theta' \vdash_l vw}$	

Fig. 7. Predicate of linear expressions with resources

C.4 Proof of (theorem 9) in the unordered case

We define the indexed predicate $M; \Gamma \vdash_l t$ by induction with M the multiset of resources and Γ the set of variables in t . Given a linear expression $M; \Gamma \vdash_l t$, we define its multiset of resources $MR(t) \stackrel{\text{def}}{=} M$. The concatenation of contexts is always defined: $\Theta = (M_1; \Gamma_1) @ \Theta' = (M_2; \Gamma_2) \stackrel{\text{def}}{=} M_1, M_2; \Gamma_1, \Gamma_2$.

We can then prove the following substitution lemma:

Lemma 17. *If $\Theta \vdash_l t$ and $\Theta', x \vdash_l u$, then $\Theta @ \Theta' \vdash_l u[t/x]$.*

Proof. By induction on u :

- $()$, **new**, **delete**, r_n : by absurdity from the induction hypothesis (IH): no variables in the context.
- $; x; \vdash_l x$: we have $\Theta \vdash_l t$ and $x[t/x] = t$, hence $\Theta \vdash_l x[t/x]$.
- $\iota_i v$: by IH we get $\Theta \vdash_l v[t/x]$ and $(\iota_i v)[t/x] = \iota_i(v[t/x])$, hence $\Theta \vdash_l (\iota_i v)[t/x]$.
- $\pi_i v$: similarly, because $(\pi_i v)[t/x] = \pi_i(v[t/x])$ the SL holds by IH.
- $\langle u_1, u_2 \rangle$: because $\langle u_1, u_2 \rangle[t/x] = \langle u_1[t/x], u_2[t/x] \rangle$, the SL hold by IH.
- $\lambda y. u_1$: because $\lambda y. (u_1[t/x]) = (\lambda y. u_1)[t/x]$, both SL hold by IH.
- (v, w) : by case analysis to split the context $\Theta @ \Theta'$, whether x is in v or w :
 - $\Theta = M_v; \Gamma_v, x$ and $\Theta' = M_w; \Gamma_w$: $(v, w)[t/x] = (v[t/x], w)$ so the SL holds by IH on $\Theta \vdash_l v$.
 - $\Theta = M_v; \Gamma_v$ and $\Theta' = M_w; \Gamma_w, x$: $(v, w)[t/x] = (v, w[t/x])$ so the SL holds by IH on $\Theta' \vdash_l w$.
- let $y = u_1$ in u_2 : similarly, we proceed by case analysis on the context:
 - $x \in \Theta$: (let $y = u_1$ in u_2)[t/x] = (let $y = u_1$ in $u_2[t/x]$) so the SL holds by IH on $\Theta, y \vdash_l u_2$.
 - $x \in \Theta'$: (let $y = u_1$ in u_2)[t/x] = (let $y = u_1[t/x]$ in u_2) so the SL holds by IH on $\Theta' \vdash_l u_1$.
- $(v, y. u_1, z. u_2)$: by case analysis on the context:
 - $x \in \Theta$: $(v, y. u_1, z. u_2)[t/x] = (v, y. u_1[t/x], z. u_2[t/x])$ so the SL holds by IH on $\Theta, y \vdash_l u_1$ and $\Theta, z \vdash_l u_2$.
 - $x \in \Theta'$: $(v, y. u_1, z. u_2)[t/x] = (v[t/x], y. u_1, z. u_2)$ the SL holds by IH on $\Theta' \vdash_l v$.
- $\delta(v, (y, z). u_1)$: by case analysis on the context.
- $\delta(v, (). t)$: by case analysis on the context.

$\frac{}{\vdash_l^S \star}$	$\frac{M_v; \vdash_l v \quad M_s \vdash_l^S s}{M_s \vdash M_v \vdash_l^S v \cdot s}$	$\frac{M_t; x \vdash_l t \quad M_s \vdash_l^S s}{M_s \vdash M_t \vdash_l^S (x^+.t) \cdot s}$
	$\frac{M_t; \vdash_l t \quad M_s \vdash_l^S s}{M_t \vdash M_s \vdash_l^C \langle t \mid s \mid l \rangle}$	

Fig. 8. Typing rules of ordered stacks and commands

- vw : by case analysis on the context.

We then extend $\vdash_l$ for stacks and commands, with a multiset of resources and no variables:

Lemma 18. *Reducing a linear command results in a linear command with the same multiset of resources, i.e. for all c_1, c_2, M such that $M \vdash_l^C c_1$ and $c_1 \rightsquigarrow c_2$ one has $M \vdash_l^C c_2$.*

Proof. By induction on reduction steps $c_1 \rightsquigarrow c_2$:

- $\langle \text{let } x^+ = t \text{ in } u \mid s \mid l \rangle \rightsquigarrow \langle t \mid (x^+.u) \cdot s \mid l \rangle$: by definition of $M \vdash_l^C c_1$, we get $M_t; \vdash_l t$, $M_u; x \vdash_l u$ and $M_s \vdash_l^S s$ with $M = M_s \vdash M_u \vdash M_t \vdash l$, hence we have $M_s \vdash M_u \vdash_l (x^+.u) \cdot s$ and so $M_s \vdash M_u \vdash M_t \vdash l = M \vdash_l^C c_2$.
- $\langle \text{let } x^- = t \text{ in } u \mid s \mid l \rangle \rightsquigarrow \langle u[t/x] \mid s \mid l \rangle$: by definition, we get $M_t; \vdash_l t$, $M_u; x \vdash_l u$ and $M_s \vdash_l^S s$, hence by SL we have $M_t \vdash M_u; \vdash_l t[u/x]$ and so $M \vdash_l^C c_2$.
- $\langle \pi_i t \mid s \mid l \rangle \rightsquigarrow \langle t \mid \pi_i \cdot s \mid l \rangle$: by definition, we get $M_t; \vdash_l t$ and $M_s \vdash_l^S s$, hence $M \vdash_l^C c_2$.
- $\langle v \mid (x.t) \cdot s \mid l \rangle \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle$: by definition, we get $M_v; \vdash_l v$, $M_t; x \vdash_l t$ and $M_s \vdash_l^S s$, hence by SL we have $M_t \vdash M_v; \vdash_l t[v/x]$ and so $M \vdash_l^C c_2$.
- $\langle \lambda x. t \mid v \cdot s \mid l \rangle \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle$: by definition, we get $M_t; x \vdash_l t$, $M_v; \vdash_l v$ and $M_s \vdash_l^S s$, hence by SL we have $M_t \vdash M_v; \vdash_l t[v/x]$ and so $M \vdash_l^C c_2$.
- $\langle \langle t_1, t_2 \rangle \mid \pi_i \cdot s \mid l \rangle \rightsquigarrow \langle t_i \mid s \mid l \rangle$: by definition, we get $M_t; \vdash_l t_1$, $M_t; \vdash_l t_2$ and $M_s \vdash_l^S s$, hence $M \vdash_l^C c_2$.
- $\langle \delta((v, w), (x, y).t) \mid s \mid l \rangle \rightsquigarrow \langle t[v/x, w/y] \mid s \mid l \rangle$: by definition, we get $M_v; \vdash_l v$, $M_w; \vdash_l w$, $M_t; x, y \vdash_l t$ and $M_s \vdash_l^S s$, hence by SL twice we have $M_t \vdash M_v \vdash M_w; \vdash_l t[v/x][w/y] = t[v/x, w/y]$ and so $M \vdash_l^C c_2$.
- $\langle \delta((), ().t) \mid s \mid l \rangle \rightsquigarrow \langle t \mid s \mid l \rangle$: by definition, we get $M_t; \vdash_l t$ and $M_s \vdash_l^S s$, hence $M \vdash_l^C c_2$.
- $\langle \delta(t_i v, x_1.t_1, x_2.t_2) \mid s \mid l \rangle \rightsquigarrow \langle t_i[x_i/v] \mid s \mid l \rangle$: by definition, we get $M_v; \vdash_l v$, $M_t; x_1 \vdash_l t_1$, $M_t; x_2 \vdash_l t_2$ and $M_s \vdash_l^S s$, hence by SL we have $M_t \vdash M_v; \vdash_l t_i[v/x_i]$ and so $M \vdash_l^C c_2$.
- $\langle \text{new} \mid () \cdot s \mid r_n :: l \rangle \rightsquigarrow \langle t_1 r_n \mid s \mid l \rangle$: by definition, we get $\vdash_l \text{new}$, $\vdash_l ()$ and $M_s \vdash_l^S s$, hence $M_s \vdash r_n :: l = M \vdash_l^C c_2$.
- $\langle \text{new} \mid () \cdot s \mid \text{Nil} \rangle \rightsquigarrow \langle t_2 () \mid s \mid \text{Nil} \rangle$: by definition, we get $\vdash_l \text{new}$, $\vdash_l ()$ and $M_s \vdash_l^S s$, hence $M_s = M \vdash_l^C c_2$.
- $\langle \text{delete} \mid r_n \cdot s \mid l \rangle \rightsquigarrow \langle () \mid s \mid r_n :: l \rangle$: by definition, we get $\vdash_l \text{delete}$, $r_n; \vdash_l r_n$ and $M_s \vdash_l^S s$, hence $M_s \vdash r_n :: l = M \vdash_l^C c_2$.

Finally, we show that $\forall \vdash t : P \in Pr, \forall v, \forall l, \forall l', \langle t \mid \star \mid l \rangle \rightsquigarrow \langle v \mid \star \mid l' \rangle \Rightarrow \exists \sigma, l' = \sigma(l)$.

Proof. By progress and subject reduction, we have that $\vdash v : W$. By induction on the typing derivation, we have that $\vdash t : A \Rightarrow \vdash_l t$, hence $l \vdash_l^C \langle t \mid \star \mid l \rangle$. By the previous lemma, we get $l \vdash_l^C \langle v \mid \star \mid l' \rangle$. Because W is central, v is a final value, so it does not contain any resource. Hence, l and l' are obtained from the same multiset, so $\exists \sigma, l' = \sigma(l)$.

D Affine language translation

D.1 Proof of well-typed translation

All terms must be translated to values. Recall that $\Downarrow A \stackrel{\text{def}}{=} A \& 1$, hence the type of translated expressions and negative values is negative, so they have no value restriction. So, we must only check the value restriction when translating positive values.

Contexts are translated by translating each type separately, so $(\Gamma, \Delta)^+ = \Gamma^+, \Delta^+$.

Values of central types are translated to the same value and type in $\mathcal{L}$, so $\llbracket \vdash \text{new_fail} : E \rrbracket = \vdash \text{new_fail} : E$. Moreover, because $\mathcal{O}_{\mathcal{E}, \text{move}}$ is an extension of $\mathcal{L}$, it can use linear swaps for exceptions.

We proceed by cases:

- $\llbracket x \vdash x : A \rrbracket$ must be a value of type $A^+ \vdash A^+$. Hence, x is well-typed.
- $\llbracket \vdash \text{drop} : 1 \multimap A \rrbracket$ must be a value of type $\vdash (1 \multimap A^+) \& 1$. Hence, $\langle \text{drop}_A, () \rangle$ is well-typed.
- $\llbracket \Gamma \vdash \text{coerc}(v) : P \rrbracket$ must be a value of type $\Gamma^+ \vdash (P^+ \oplus E) \& I$. By IH, we have $\Gamma^+ \vdash \llbracket v \rrbracket : P^+$. Hence, $\langle t_1 \llbracket v \rrbracket, \text{drop_ctx}_\Gamma \rangle$ is well-typed.
- $\llbracket \Gamma, x : A, y : B, \Delta \vdash \text{move}(x, y) \text{ in } t : C \rrbracket$ must be a value of type $\Gamma^+, y : B^+, x : A^+, \Delta^+ \vdash \Downarrow C^-$. By IH, we have $\Gamma^+, x : A^+, y : B^+, \Delta^+ \vdash \llbracket t \rrbracket : \Downarrow C^-$. It's the only rule where we allow ourselves to use $\mathcal{L}$ structural rule to swap premises. Hence, $\llbracket t \rrbracket$ is well-typed.
- $\llbracket \vdash \text{new} : R \oplus E \multimap 1 \rrbracket$ must be a value of type $\vdash R \oplus E \multimap 1$. We have $\vdash \text{new_fail} : E$, so let $x = \text{new}()$ in $\delta(x, r, t_1 r, i, i; t_2 \text{new_fail})$ is well-typed.
- $\llbracket \Gamma, \Delta \vdash \text{let } x = (v : A) \text{ in } t : B \rrbracket$ must be a value of type $\Gamma^+, \Delta^+ \vdash \Downarrow B^-$. By IH, we have $\Delta^+ \vdash \llbracket v \rrbracket : A^+$ and $\Gamma^+, x : A^+ \vdash \llbracket t \rrbracket : \Downarrow B^-$. Hence, let $x = \llbracket t \rrbracket$ in $\llbracket u \rrbracket$ is well-typed.
- $\llbracket \Gamma, \Delta \vdash \text{let } x = (t : P) \text{ in } u : A \rrbracket$ must be a value of type $\Gamma^+, \Delta^+ \vdash A^- \& I$. By IH, we have $\Delta^+ \vdash \llbracket t \rrbracket : (P^+ \oplus E) \& I$ and $\Gamma^+, x : P^+ \vdash \llbracket u \rrbracket : A^- \& I$. Then, $\text{raise}_{\Gamma}^{A \& I}(e)$ is of type $\Gamma^+ \vdash A^- \& I$, so let $s = \pi_1 \llbracket t \rrbracket$ in $\delta(s, x, \llbracket u \rrbracket, e, \text{raise}_{\Gamma}^{A \& I}(e))$ is well-typed.
- $\llbracket \vdash \text{raise} : A \multimap E \rrbracket$ must be a value of type $\vdash (A^- \multimap E) \& I$. Hence, $\langle \lambda e. \text{raise}_{\star}^A(e), () \rangle$ is well-typed.
- $\llbracket \Gamma, \Delta \vdash \text{try } x \Leftarrow (t : P) \text{ in } u \text{ unless } e \Rightarrow u' : B \rrbracket$ must be a value of type $\Gamma^+, \Delta^+ \vdash \Downarrow B^-$. By IH, we have:
 - $\Delta^+ \vdash \llbracket t \rrbracket : (P^+ \oplus E) \& I$.
 - $\Gamma^+, x : P^+ \vdash \llbracket u \rrbracket : \Downarrow B^-$.

- $\Gamma^+, e : E \vdash \llbracket u' \rrbracket : \Downarrow B^-$.
Hence, let $s = \pi_1 \llbracket t \rrbracket$ in $\delta(s, x. \llbracket u \rrbracket, e. \llbracket u' \rrbracket)$ is well-typed.
- $\llbracket \Gamma, \Delta \vdash (v, w) : A \otimes B \rrbracket$ must be a value of type $\Gamma^+, \Delta^+ \vdash A^+ \otimes B^+$. By IH, we have $\Gamma^+ \vdash \llbracket v \rrbracket : A^+$ and $\Delta^+ \vdash \llbracket w \rrbracket : B^+$. Hence, $(\llbracket v \rrbracket, \llbracket w \rrbracket)$ is well-typed.
- $\llbracket \Gamma, \Delta, \Gamma' \vdash \delta(v, (x, y).t) : C \rrbracket$ must be a value of type $\Gamma^+, \Delta^+, \Gamma'^+ \vdash \Downarrow C^-$. By IH, we have $\Delta^+ \vdash \llbracket v \rrbracket : A^+ \otimes B^+$ and $\Gamma^+, x : A^+, y : B^+, \Gamma'^+ \vdash \llbracket t \rrbracket : \Downarrow C^-$. Hence, $\delta(\llbracket v \rrbracket, (x, y). \llbracket t \rrbracket)$ is well-typed.
- $\llbracket () : 1 \rrbracket$ must be a value of type $\vdash 1$. Hence, $()$ is well-typed.
- $\llbracket \Gamma, \Delta, \Gamma' \vdash \delta(v, ().t) : C \rrbracket$ must be a value of type $\Gamma^+, \Delta^+, \Gamma'^+ \vdash \Downarrow C^-$. By IH, we have $\Delta^+ \vdash \llbracket v \rrbracket : 1$ and $\Gamma^+, \Gamma'^+ \vdash \llbracket t \rrbracket : \Downarrow C^-$. Hence, $\delta(\llbracket v \rrbracket, (). \llbracket t \rrbracket)$ is well-typed.
- $\llbracket \Gamma \vdash \iota_1 v : A \oplus B \rrbracket$ must be a value of type $\Gamma^+ \vdash A^+ \oplus B^+$. By IH, we have $\Gamma^+ \vdash \llbracket v \rrbracket : A^+$. Hence, $\iota_1 \llbracket v \rrbracket$ is well-typed.
- $\llbracket \Gamma \vdash \iota_2 v : A \oplus B \rrbracket$ must be a value of type $\Gamma^+ \vdash A^+ \oplus B^+$. By IH, we have $\Gamma^+ \vdash \llbracket v \rrbracket : B^+$. Hence, $\iota_2 \llbracket v \rrbracket$ is well-typed.
- $\llbracket \Gamma, \Delta, \Gamma' \vdash \delta(v, x.t, y.u) : C \rrbracket$ must be a value of type $\Gamma^+, \Delta^+, \Gamma'^+ \vdash \Downarrow C^-$. By IH, we have:
 - $\Delta^+ \vdash \llbracket v \rrbracket : A^+ \oplus B^+$.
 - $\Gamma^+, x : A^+, \Gamma'^+ \vdash \llbracket t \rrbracket : \Downarrow C^-$.
 - $\Gamma^+, y : B^+, \Gamma'^+ \vdash \llbracket u \rrbracket : \Downarrow C^-$.
 Hence, $\delta(\llbracket v \rrbracket, x. \llbracket t \rrbracket, y. \llbracket u \rrbracket)$ is well-typed.
- $\llbracket \Gamma \vdash \lambda x. t : B \multimap A \rrbracket$ must be a value of type $\Gamma^+ \vdash (B^- \multimap A^+) \& I$. By IH, we have $x : A^+, \Gamma^+ \vdash \llbracket t \rrbracket : B^- \& I$. Hence, $\langle \lambda x. \pi_1 \llbracket t \rrbracket, \text{drop_ctx}_\Gamma \rangle$ is well-typed.
- $\llbracket \Gamma, \Delta \vdash vw : B \rrbracket$ must be a value of type $\Gamma^+, \Delta^+ \vdash \Downarrow B^-$. By IH, we have $\Gamma^+ \vdash \llbracket w \rrbracket : A^+$ and $\Gamma^+ \vdash \llbracket v \rrbracket : (B^- \multimap A^+) \& I$. Hence, $\langle (\pi_1 \llbracket v \rrbracket) \llbracket w \rrbracket, \text{drop_ctx}_{\Gamma, \Delta} \rangle$ is well-typed.
- $\llbracket \Gamma \vdash \langle t, u \rangle : A \& B \rrbracket$ must be a value of type $\Gamma^+ \vdash (A^- \& B^-) \& I$. By IH, we have $\Gamma^+ \vdash \llbracket t \rrbracket : A^- \& I$ and $\Gamma^+ \vdash \llbracket u \rrbracket : B^- \& I$. Hence, $\langle \langle \pi_1 \llbracket t \rrbracket, \pi_1 \llbracket u \rrbracket \rangle, \text{drop_ctx}_\Gamma \rangle$ is well-typed.
- $\llbracket \Gamma \vdash \pi_1 v : A \rrbracket$ must be a value of type $\Gamma^+ \vdash A^- \& I$. By IH, we have $\Gamma^+ \vdash \llbracket v \rrbracket : (A^- \& B^-) \& I$. Hence, $\langle \pi_1 \llbracket v \rrbracket, \text{drop_ctx}_\Gamma \rangle$ is well-typed.
- $\llbracket \Gamma \vdash \pi_2 v : B \rrbracket$ must be a value of type $\Gamma^+ \vdash B^- \& I$. By IH, we have $\Gamma^+ \vdash \llbracket v \rrbracket : (A^- \& B^-) \& I$. Hence, $\langle \pi_2 \llbracket v \rrbracket, \text{drop_ctx}_\Gamma \rangle$ is well-typed.

E Categorical semantics

E.1 Interpretation of types and terms

Resources appearing in terms of the operational semantics are interpreted as variables of type R .

Each type has a positive and negative interpretation:

- $1^+ \stackrel{\text{def}}{=} I, (A \otimes B)^+ \stackrel{\text{def}}{=} A^+ \otimes B^+, (A \oplus B)^+ \stackrel{\text{def}}{=} A^+ \oplus B^+, N^+ \stackrel{\text{def}}{=} G(N^-)$
- $(B \multimap A)^- \stackrel{\text{def}}{=} B^- \multimap A^+, (A \& B)^- \stackrel{\text{def}}{=} A^- \times B^-, P^- \stackrel{\text{def}}{=} F(P^+)$

A context with resources $\Gamma; [r_1 \dots r_n]; \Delta$ is interpreted positively by $\Gamma^+ \otimes R^n \otimes \Delta^+$ with $R^{n+1} \stackrel{\text{def}}{=} R^n \otimes R, R^0 \stackrel{\text{def}}{=} I, (x : A, \Gamma)^+ \stackrel{\text{def}}{=} A^+ \otimes \Gamma^+$. Lists of resources are then written L_x and their translation L_x^+ .

Each judgement kind with resources have an interpretation:

- Values $\Theta \vdash_o v : A$ in $\mathcal{L}(\Theta^+, A^+)$.
- Expressions $\Theta \vdash_o t : A$ in $\mathcal{L}(\Theta^+, GA^-)$.
- Stacks $Ls; s : A \vdash_o B$ in $\mathcal{L}(Ls^+ \otimes A^-, B^-)$.
- Commands $Lc \vdash_o c : A$ in $\mathcal{L}(Lc^+, A^-)$.

To interpret terms, types in the derivation of $\vdash_o$ are made explicit. In particular, stacks $L \vdash_o^S s$ are written $L; s : A \vdash_o^S B$.

Composition is written in the diagrammatic order with $;$. η is the unit of GF . f^* is the adjoint of f under $F \dashv G$. f^{str_Γ} precomposes f with the isomorphism $\Gamma \otimes FA \simeq F(\Gamma \otimes A)$ in the relevant direction. $f^{\lambda^-/-\circ}$ is the adjoint of f under $\otimes \dashv \circ$ (respectively $\circ$), and $ev^{\circ/-\circ}$ the evaluation morphism of this adjunction. dis is the canonical morphism $\Gamma \otimes (A \oplus B) \otimes \Gamma' \rightarrow (\Gamma \otimes A \otimes \Gamma') \oplus (\Gamma \otimes B \otimes \Gamma')$ obtained from the fact that the category is monoidal distributive, because it is biclosed. Apart from str , associators and unitors of the monoidal product are left implicit.

Values have explicit coercions inserted on positive values where expressions are expected, written $coerc(v)$ below.

- $\llbracket \Theta \vdash_o coerc(v) : P \rrbracket \stackrel{\text{def}}{=} \llbracket v \rrbracket; \eta : \mathcal{L}(\Theta^+, GFP^+)$
- $\llbracket x : A \vdash_o x : A \rrbracket \stackrel{\text{def}}{=} id_{A^+} : \mathcal{L}(A^+, A^+)$
- $\llbracket r : R \vdash_o r : R \rrbracket \stackrel{\text{def}}{=} id_R : \mathcal{L}(R, R)$
- $\llbracket \vdash_o () : 1 \rrbracket \stackrel{\text{def}}{=} id_I : \mathcal{L}(I, I)$
- $\llbracket \Theta, \Theta' \vdash_o (v, w) : A \otimes B \rrbracket \stackrel{\text{def}}{=} \llbracket v \rrbracket \otimes \llbracket w \rrbracket : \mathcal{L}(\Theta^+ \otimes \Theta'^+, A^+ \otimes B^+)$
- $\llbracket \Theta \vdash_o \iota_1 v : A \oplus B \rrbracket \stackrel{\text{def}}{=} \llbracket v \rrbracket; inl : \mathcal{L}(\Theta^+, A^+ \oplus B^+)$
- $\llbracket \Theta \vdash_o \iota_2 v : A \oplus B \rrbracket \stackrel{\text{def}}{=} \llbracket v \rrbracket; inr : \mathcal{L}(\Theta^+, A^+ \oplus B^+)$
- $\llbracket \Theta, \Theta', \Theta'' \vdash_o \delta(v, (), t) : C \rrbracket \stackrel{\text{def}}{=} \llbracket t \rrbracket : \mathcal{L}(\Theta^+ \otimes \Theta'^+ \otimes \Theta'', GC^-)$
- $\llbracket \Theta, \Theta', \Theta'' \vdash_o \delta(v, (x, y), t) : C \rrbracket \stackrel{\text{def}}{=} (id_{\Theta^+} \otimes \llbracket v \rrbracket \otimes id_{\Theta''^+}); \llbracket t \rrbracket : \mathcal{L}(\Theta^+ \otimes \Theta'^+ \otimes \Theta'', GC^-)$
- $\llbracket \Theta, \Theta', \Theta'' \vdash_o \delta(v, x.t, y.u) : C \rrbracket \stackrel{\text{def}}{=} (id_{\Theta^+} \otimes \llbracket v \rrbracket \otimes id_{\Theta''^+}); dis; (\llbracket t \rrbracket \mid \llbracket u \rrbracket) : \mathcal{L}(\Theta^+ \otimes \Theta'^+ \otimes \Theta'', GC^-)$
- $\llbracket \Theta \vdash_o \lambda x. t : B \multimap A \rrbracket \stackrel{\text{def}}{=} \llbracket t \rrbracket^*; str_{\Theta^+}; \lambda^-; * : \mathcal{L}(\Theta^+, G(B^- \multimap A^+))$
- $\llbracket \Theta \vdash_o \langle t, u \rangle \rrbracket \stackrel{\text{def}}{=} (\llbracket t \rrbracket^* \times \llbracket u \rrbracket^*)^* : \mathcal{L}(\Theta^+, G(A^- \& B^-))$
- $\llbracket \Theta, \Theta' \vdash_o vw : B \rrbracket \stackrel{\text{def}}{=} ((\llbracket w \rrbracket \otimes \llbracket v \rrbracket^*); ev)^{str_{\Theta^+}; *} : \mathcal{L}(\Theta^+ \otimes \Theta'^+, GB^-)$
- $\llbracket \Theta \vdash_o \pi_1 v : A \rrbracket \stackrel{\text{def}}{=} (v^*; \pi_1)^* : \mathcal{L}(\Theta^+, GA^-)$
- $\llbracket \Theta \vdash_o \pi_2 v : B \rrbracket \stackrel{\text{def}}{=} (v^*; \pi_2)^* : \mathcal{L}(\Theta^+, GB^-)$
- $\llbracket \Theta, \Theta' \vdash_o \text{let } x = (t : P) \text{ in } u \rrbracket \stackrel{\text{def}}{=} ((id_{\Theta^+} \otimes \llbracket t \rrbracket^*); \llbracket u \rrbracket^*; str_{\Theta^+})^{str_{\Theta^+}; *} : \mathcal{L}(\Theta^+ \otimes \Theta'^+, GB^-)$
- $\llbracket \Theta, \Theta' \vdash_o \text{let } x = (t : N) \text{ in } u \rrbracket \stackrel{\text{def}}{=} (id_{\Theta^+} \otimes \llbracket t \rrbracket); \llbracket u \rrbracket : \mathcal{L}(\Theta^+ \otimes \Theta'^+, GB^-)$
- $\llbracket \vdash_o \text{new} : R \oplus 1 \multimap 1 \rrbracket \stackrel{\text{def}}{=} l \mapsto i \mapsto \text{match } l \{ x :: t \mapsto (inr(x), t) \mid [] \mapsto (inr(), []) \} : \mathcal{L}(I, G(F(R \oplus I) \multimap I))$

Stacks are interpreted as such:

- $\llbracket []; \star : A \vdash_o A \rrbracket \stackrel{\text{def}}{=} id_{A^-} : \mathcal{L}(A^-, A^-)$
- $\llbracket Ls; \pi_1 \cdot s : A \& B \vdash_o C \rrbracket \stackrel{\text{def}}{=} \pi_1; \llbracket s \rrbracket : \mathcal{L}(Ls^+ \otimes (A^- \& B^-), C^-)$
- $\llbracket Ls; \pi_2 \cdot s : A \& B \vdash_o C \rrbracket \stackrel{\text{def}}{=} \pi_2; \llbracket s \rrbracket : \mathcal{L}(Ls^+ \otimes (A^- \& B^-), C^-)$
- $\llbracket Ls \dashv Lt; v \cdot s : B \multimap A \vdash_o C \rrbracket \stackrel{\text{def}}{=} (id_{Ls^+} \otimes ((\llbracket v \rrbracket \otimes id_{B^- \multimap A^+}); ev^{\circ})); \llbracket s \rrbracket : \mathcal{L}(Ls^+ \otimes Lt^+ \otimes (B^- \multimap A^+), C^-)$

- $\llbracket Ls \# Lt; (x^+.t) \cdot s : A \vdash_o C \rrbracket \stackrel{\text{def}}{=} (id_{Ls^+} \otimes \llbracket t \rrbracket^{*,str_{Lt^+}}; \llbracket s \rrbracket : \mathcal{L}(Ls^+ \otimes Lt^+ \otimes FA^+, C^-)$

Finally, $\llbracket Ls \# Lt \# Ll \vdash_o \langle t \mid s \mid l \rangle \rrbracket \stackrel{\text{def}}{=} (((id_{Ls^+} \otimes \llbracket t \rrbracket^*); \llbracket s \rrbracket)^{str_{Ls^+};*} \otimes (! : Ll \rightarrow [R])); ev^-$.

E.2 Proof of (lemma 11)

Proof. By induction on the derivation of t with explicit coercions. Uses naturality of $\otimes$ associativity left implicit, as well as other transformations $(*, str, \times, \lambda)$ that interpret rules of negative connectives. For $*$, it means in particular that $f; (g^*; h)^* = (Ff; (g^*; h))^* = ((f; g)^*; h)^*$:

- $\llbracket \text{coerc}(w)[v/x] \rrbracket = \llbracket w[v/x] \rrbracket; \eta = \llbracket \text{coerc}(w[v/x]) \rrbracket$ by induction hypothesis.
- $\llbracket x[v/x] \rrbracket = \llbracket v \rrbracket$.
- No variables in r , **new** nor $()$.
- $\llbracket \Gamma, \Delta \vdash_o (v, w) \rrbracket$: either $x \in \Gamma$ or $x \in \Delta$, in which cases we need to show $\llbracket (w, w')[v/x] \rrbracket = \llbracket (w[v/x], w') \rrbracket$ or $\llbracket (w[v/x], w') \rrbracket$. Those hold by IH.
- $\llbracket (l_i w)[v/x] \rrbracket = \llbracket l_i(w[v/x]) \rrbracket$ by IH.
- $\llbracket \delta(w, ().t)[v/x] \rrbracket$: split case on $v \in \Gamma, v \in \Delta$ or $v \in \Gamma'$. In each case, the SL holds by IH.
- $\llbracket \delta(w, (y, z).t)[v/x] \rrbracket$: same split case. In each case, the SL holds by IH.
- $\llbracket \delta(w, y.t, z.u)[v/x] \rrbracket$: same split case. In each case, the SL holds by IH.
- $\langle t, u \rangle$: for $\langle t, u \rangle[v/x] = \langle t[v/x], u[v/x] \rangle$, we need to show that $(id_{\Gamma^+} \otimes \llbracket v \rrbracket \otimes id_{\Gamma'^+}); (\llbracket t \rrbracket^* \times \llbracket u \rrbracket^*)^* = (((id_{\Gamma^+} \otimes \llbracket v \rrbracket \otimes id_{\Gamma'^+}); \llbracket t \rrbracket^*) \times ((id_{\Gamma^+} \otimes \llbracket v \rrbracket \otimes id_{\Gamma'^+}); \llbracket u \rrbracket^*))^*$. Hence, the SL holds by IH and naturality of $*$ and $\times$.
- $\pi_i v$: similarly, because $(\pi_i v)[t/x] = \pi_i(v[t/x])$ the SL holds by IH and naturality of $*$.
- $\lambda y. u_1$: because $\lambda y. (u_1[t/x]) = (\lambda y. u_1)[t/x]$, both SL hold by IH and naturality of $*$ and str .
- ww' : split case on whether we have $w[v/x] = w$ or $w'[v/x] = w'$, in both cases it holds by IH and naturality of $*$, str and λ .
- $\llbracket \Gamma, \Delta \vdash_o (\text{let } y = (t : P) \text{ in } u)[v/x] \rrbracket$: split case on $v \in \Gamma$ or $v \in \Delta$, in both cases it holds by SL and naturality of $*$ and str .
- $\llbracket \Gamma, \Delta \vdash_o (\text{let } y = (t : N) \text{ in } u)[v/x] \rrbracket$: same split case. In both cases it holds by SL.

E.3 Proof of (theorem 12)

Proof. By case analysis on reduction steps: when only t changes, we show that expressions from both commands are equal. Otherwise, when the stack also changes, we show that $(Ls^+ \otimes \llbracket t \rrbracket^*); \llbracket s \rrbracket$ from both command are equal. Finally, we consider the whole command interpretation pointwise for **new** and **delete** reduction rules.

- $\langle \text{let } x^- = v \text{ in } u \mid s \mid l \rangle \rightsquigarrow \langle u[t/x] \mid s \mid l \rangle$: we need to show that $\llbracket \text{let } x^- = v \text{ in } u \rrbracket = \llbracket u[t/x] \rrbracket$. Both expressions are equal by SL and definition of binding a negative value, that is precomposition.

- $\langle \text{let } x^+ = t \text{ in } u \mid s \mid l \rangle \rightsquigarrow \langle t \mid (x^+.u) \cdot s \mid l \rangle$: we need to show that $(id_{L_{S^+}} \otimes ((id_{L_{U^+}} \otimes \llbracket t \rrbracket^*); \llbracket u \rrbracket^{*,str_{L_{U^+}}})^{str_{L_{U^+}}}; \llbracket s \rrbracket) = (id_{L_{S^+}} \otimes id_{L_{U^+}} \otimes \llbracket t \rrbracket^*); (id_{L_{S^+}} \otimes \llbracket u \rrbracket^{*,str_{L_{U^+}}}); \llbracket s \rrbracket$. It holds after cancelling the two $*$, re-associating parentheses with the first str of the left expression then distributing $\otimes$.
- $\langle vw \mid s \mid l \rangle \rightsquigarrow \langle v \mid w \cdot s \mid l \rangle$: holds after cancelling $*$; $*$.
- $\langle \pi_i t \mid s \mid l \rangle \rightsquigarrow \langle t \mid \pi_i \cdot s \mid l \rangle$: holds after cancelling $*$; $*$ on the left expression.
- $\langle v \mid (x.t) \cdot s \mid l \rangle \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle$: $\llbracket \text{coerc}(v) \rrbracket^* = F[\llbracket v \rrbracket]$, then it holds by SL.
- $\langle \lambda x. t \mid v \cdot s \mid l \rangle \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle$: holds after cancelling $*$; $*$ and computing λ with ev on the left expression, then applying SL.
- $\langle \langle t_1, t_2 \rangle \mid \pi_i \cdot s \mid l \rangle \rightsquigarrow \langle t_i \mid s \mid l \rangle$: holds after cancelling $*$; $*$ and computing π_i on the left expression.
- $\langle \delta((v, w), (x, y).t) \mid s \mid l \rangle \rightsquigarrow \langle t[v/x, w/y] \mid s \mid l \rangle$: both command expressions are equal by applying SL twice.
- $\langle \delta((), ().t) \mid s \mid l \rangle \rightsquigarrow \langle t \mid s \mid l \rangle$: both command expressions are equal by unit laws.
- $\langle \delta(\iota_i v, x_1.t_1, x_2.t_2) \mid s \mid l \rangle \rightsquigarrow \langle t_i[x_i/v] \mid s \mid l \rangle$: the left expression with SL is equal to the right after computing the distribution morphism and inl/inr (for each i).
- $\langle \text{new} \mid () \cdot s \mid r_n :: l \rangle \rightsquigarrow \langle \iota_1 r_n \mid s \mid l \rangle$: by definition of popping elements from a non-empty list of resources.
- $\langle \text{new} \mid () \cdot s \mid \text{Nil} \rangle \rightsquigarrow \langle \iota_2 () \mid s \mid \text{Nil} \rangle$: by definition of popping elements from an empty list of resources.
- $\langle \text{delete} \mid r_n \cdot s \mid l \rangle \rightsquigarrow \langle () \mid s \mid r_n :: l \rangle$: by definition of pushing an element in a list of resources.

E.4 Proof that center coincides with resource-free objects, and objects with trivial destructors

We will show the bi-implications $1 \iff 2$ and $2 \iff 3$ for the following properties on object $E \in \mathcal{L}$ (see proposition 13):

- 1 E is in the Drinfeld center of $\mathcal{L}$.
- 2 For all l such that $l \neq []$, $E(l) = \emptyset$.
- 3 There is a morphism $\mathcal{L}(E, I)$.

Recall that a morphism $\mathcal{L}(A, B)$ is a function family $\forall l, A(l) \rightarrow B(l)$.

(2 $\implies$ 3) Resource-free objects have a trivial destructor:

We define the family of functions $d : \forall l, E(l) \rightarrow I(l)$ by case on l :

- If $l = []$, then $I([]) = \{\star\}$ so we pick the unique function to it.
- Otherwise, $E(l) = \emptyset$ so we pick the unique function from it.

(3 $\implies$ 2) Objects with a trivial destructor have no resources:

There is a function family $\forall l, E(l) \rightarrow I(l)$. If $l = []$, $I(l) = \emptyset$ so it must be the case that $E(l) = \emptyset$.

(1 $\implies$ 2) Objects in the Drinfeld center have no resources. Such objects are pairs $E : \mathcal{L}, \theta : E \otimes - \simeq - \otimes E$ such that

$$\theta_{A \otimes B} = (\theta_A \otimes id); (id \otimes \theta_B) .$$

So, we have a bijection family

$$\forall A, \forall l, \forall j \vdash k = l, \exists j' \vdash k' = l, E(j) \times A(k) \simeq A(j') \times E(k') .$$

Suppose that there exists an inhabited list $r :: t$ such that $E(r :: t)$ is inhabited. We will show that no such bijection family exists.

We fix a resource $r' \neq r$. We instantiate θ with $A(t) \stackrel{\text{def}}{=} \{\star\}$, $A(-) \stackrel{\text{def}}{=} \emptyset$, $j \stackrel{\text{def}}{=} r :: t$, $k \stackrel{\text{def}}{=} [r']$. We must show that for all j', k' such that $j' \vdash k' = r :: t \vdash [r']$, there is no function $E(r :: t) \times A([r']) \rightarrow A(j') \times E(k')$: because $E(r :: t)$ and $A([r'])$ are inhabited, $A(j') \times E(k')$ needs to be inhabited. However, j' first element must be r , so $j' \neq [r']$. Hence $A(j') = \emptyset$, so the product is empty.

(2 $\Rightarrow$ 1) We show that resource-free objects are in the Drinfeld center. Given E resource-free, we define $\theta : \forall A, \forall l, \forall j \vdash k = l, \exists j' \vdash k' = l, E(j) \times A(k) \simeq A(j') \times E(k')$ by case on j :

- If $j = []$ then $k = l$, so we pick $j' = k, k' = []$ and the function

$$\text{swap}_{E([], A(k))} : E([]) \times A(k) \simeq A(k) \times E([])$$

- Otherwise $E(j) = \emptyset$, so we pick $j' \stackrel{\text{def}}{=} j, k' \stackrel{\text{def}}{=} k$ and the unique function from $E(j) \times A(k)$.

We are left to check naturality and the center additional condition. If $j \neq []$, they hold by universal property, so we are left with the first case. swap is natural in both arguments, so it only remains to show that

$$\theta_{A \otimes B} = (\theta_A \otimes id); (id \otimes \theta_B) .$$

Those morphisms are function families

$$\forall l, \forall i \vdash j \vdash k = l, \exists i' \vdash j' \vdash k' = l, E(i) \times A(j) \times B(k) \simeq A(i') \times B(j') \times E(k')$$

. We need to check that they are pointwise equal. Given $f : A \rightarrow B$ and $g : C \rightarrow D$, the Day convolution morphism

$$f \otimes g : \forall l, \forall j \vdash k = l, \exists j' \vdash k' = l, A(j) \times C(k) \rightarrow B(j') \times D(k')$$

equals $f(j) \times g(k)$ by picking $j' \stackrel{\text{def}}{=} j, k' \stackrel{\text{def}}{=} k$. By definition, we thus have

$$\begin{aligned} & (\theta_A \otimes id); (id \otimes \theta_B)(l = i \vdash j \vdash k) \\ &= (\text{swap}_{E(i), A(j)} \times id_{B(k)}); (id_{A(j)} \times \text{swap}_{E(i), B(k)}) \\ &= \text{swap}_{E(i), A(j) \times B(k)} \\ &= \theta_{A \otimes B}(l = i \vdash j \vdash k) . \end{aligned}$$