Linear Effects, Exceptions, and Resource Safety

A Curry-Howard Correspondence for Destructors*

Sidney Congard[1,2], Guillaume Munch-Maccagnoni[1], and Rémi Douence[2]

[1] INRIA, LS2N CNRS, Nantes, France
[2] IMT Atlantique, Nantes, France

January 25, 2026

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

We then integrate exceptions with linearity and effects by adjoining default destruction actions to types, as inspired by C++/Rust destructors. We see destructors as objects $\delta : A \to TI$ in the slice category over TI. This construction gives rise to a second calculus, the *resource call-by-push-value*, featuring exceptions and destructors, and whose weakening and exchange rules perform side-effects. It is therefore affine at the level of types but ordered at the level of derivations. As in C++ and Rust, a "move" operation—the side-effecting exchange rule—is necessary for releasing resources in random order, as opposed to LIFO order.

1 Introduction

The application of monads to study effects in programming languages [38, 39, 15, 49, 46, 31, 30], as well as the application of linearity to study resource-sensitive aspects of computation [19, 29, 3, 14, 33, 5, 26], are well-established. However, the combination of effects and resources, despite receiving some attention [50, 22, 23, 51, 36, 37, 11], 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 $\mathscr{C}$, together with families of maps

$$\eta_A : A \to TA \qquad \text{and} \qquad \mu_A : TTA \to TA \tag{1}$$

* Slightly longer version with more details of a paper that appeared in ESOP 2026 with the same title (https://doi.org/10.1007/978-3-032-22720-1_8). To cite this version: https://doi.org/10.48550/arXiv.2510.23517.

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

$$\sigma_{\Gamma,A} : \Gamma \times TA \to T(\Gamma \times A) \tag{2}$$

natural in $\Gamma \in \mathscr{C}$ and $A \in \mathscr{C}$ 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:

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

which reflects in their interpretation as morphisms in the Kleisli category of T

$$[\![t]\!] : \Gamma \to TA \qquad \text{and} \qquad [\![u]\!] : \Gamma \times A \to TB$$

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

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

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 $(\mathscr{C}, \times, 1)$ to a symmetric monoidal category $(\mathscr{C}, \otimes, I)$. This modification gives control over duplication and erasure (corresponding to contraction and weakening in logic) since the maps

$$A \to A \otimes A \qquad \text{and} \qquad A \to I \tag{5}$$

are no longer available for a general A. The linear logic viewpoint nevertheless subsumes the cartesian viewpoint using a resource modality "!". Such a resource modality is a comonad such that its Eilenberg-Moore category $\mathscr{C}^!$, whose objects are coalgebras $(A, \tau : A \to !A)$, has a symmetric monoidal structure that coincides with the symmetric monoidal structure of $\mathscr{C}$ in terms of the underlying objects, and such that this symmetric monoidal structure on $\mathscr{C}^!$ is cartesian [34, 35]. Concretely, duplication and erasure (5) are available whenever A is given with a map $\tau : A \to !A$ 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

$$A \otimes B \to B \otimes A \tag{6}$$

is no longer available unconditionally. The corresponding logics are called *ordered* instead of linear. But again, the exchange rule can be reintroduced selectively with a resource modality [21]. Ordered logic can be useful to model the order in the release of resources [47, 48, 58], as we will see again in this paper.

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 $\mathscr{C}$ and a strength now given as a family of maps

$$\sigma_{\Gamma,A} : \Gamma \otimes TA \to T(\Gamma \otimes A) \tag{7}$$

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 $S \in \mathscr{C}$ in any symmetric monoidal closed category $\mathscr{C}$. This transposition to the linear context of computational effects is well-behaved and works similarly to (non-linear) monadic effects [49, 22].

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 [8, 22, 23]:

- the monad "?" of *linearly-defined continuations* (but not *linearly-used*) $?A \stackrel{\text{def}}{=} !(A \multimap \bot) \multimap \bot$, 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 with $\oplus$ the categorical coproduct—the focus of this paper.

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 $\tau : \Gamma \to !\Gamma$, so that it can be erased. It so happens [8, 22, 23] that the monad ? of linearly-defined continuations and the exception monad $\mathcal{E}$ each do have a *(left) strength with respect to* $U^!$, where $U^! : \mathscr{C}^! \to \mathscr{C}$ is the forgetful functor of the category of coalgebras,[3] defined as a family of maps

$$\sigma_{\Gamma,A} : U^!\Gamma \otimes TA \to T(U^!\Gamma \otimes A) \tag{8}$$

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

$$U^!\Gamma \otimes A \to (U^!\Gamma \otimes A) \oplus E \qquad \text{and} \qquad U^!\Gamma \otimes E \to (U^!\Gamma \otimes A) \oplus E \tag{9}$$

corresponding respectively to the *normal case* and the *exceptional case* describing the propagation of the exception, whose copairing must be subject to the mentioned naturality and coherence conditions. The normal case is given by the left inclusion whereas the exceptional case is obtained from $U^!\Gamma \otimes E \to E$, which follows from the erasure (5) given the hypothesis that the monoidal structure on $\mathscr{C}^!$ 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 $\mathscr{C}^!$ is semi-cartesian suffices (the monoidal unit of $\mathscr{C}^!$ 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 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 in Γ 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 (see section 6).

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 designers of the C++ programming language [28, 52, 24]. (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 $\mathscr{C}$ together with a given notion of linear effect, that is, a given monad T on $\mathscr{C}$ with a strength (7). As we have explained, this is a notion of linear model with linear effect which has previously been studied [49, 22, 36]; in particular a concrete model of computation is given by a *linear call-by-push-value* (linear CBPV) calculus [11] that will provide the basis of a first linear calculus we study.

In this model, we are interested in integrating exceptions. Assuming $\mathscr{C}$ 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 strong 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 $\mathcal{E}T \to T\mathcal{E}$. 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 $A \in \mathscr{C}$ something like

$$\Gamma \otimes T(A \oplus E) \to T((\Gamma \otimes A) \oplus E) \tag{10}$$

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

$$\Gamma \otimes A \to T((\Gamma \otimes A) \oplus E) \qquad \text{and} \qquad \Gamma \otimes E \to T((\Gamma \otimes A) \oplus E)$$

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. This second type 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 $\Gamma \to TI$. 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 $\delta : \Gamma \to TI$, that is to say an object in the slice category $\mathscr{C}_{/TI}$. Recall that it is the category whose objects are arbitrary pairs $(A \in \mathscr{C}, \delta : A \to TI)$, and whose morphisms are morphisms in $\mathscr{C}$ that preserve the second component δ. The slice category $\mathscr{C}_{/TI}$ enjoys a series of nice properties, the most striking one being that it gives rise to a resource modality on $\mathscr{C}$ [10]. Indeed:

- $\mathscr{C}_{/TI}$ 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 $(I, \eta_I : I \to TI)$ and, for $(A, \delta_A : A \to TI)$ and $(B, \delta_B : B \to TI)$, the tensor product is given by:

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

  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 $U : \mathscr{C}_{/TI} \to \mathscr{C}$, that sends the monoidal structure of $\mathscr{C}_{/TI}$ to the one of $\mathscr{C}$ (strictly so).
- If we assume that $\mathscr{C}$ has finite products that we note $(\&, \top)$, then U has a right adjoint $G : \mathscr{C} \to \mathscr{C}_{/TI}$ sending objects $A \in \mathscr{C}$ to

  $$(A \,\&\, TI, \pi_2 : A \,\&\, TI \to TI) \in \mathscr{C}_{/TI} \tag{11}$$

  and morphisms $f : A \to B$ to $f \,\&\, TI$. In particular if we note $\mathcal{D} \stackrel{\text{def}}{=} UG$ the comonad on $\mathscr{C}$ associated to the adjunction, one has $\mathcal{D} = (- \,\&\, TI)$ as our resource modality.
- The latter adjunction is comonadic: $\mathscr{C}_{/TI} \simeq \mathscr{C}^{\mathcal{D}}$. This can be observed from the fact that a coalgebra $(A, \tau : A \to \mathcal{D}A)$ for $\mathcal{D}$ boils down to an object A provided with a morphism $A \to TI$ in $\mathscr{C}$ without any other condition.

This setup was advocated as a starting point to study C++ destructors in [10].

At this stage we find it useful to sum up the assumptions on $\mathscr{C}$: 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, $\mathcal{D} = (- \,\&\, TI)$ has the structure of a resource modality whose Eilenberg-Moore category is (equivalent to) $\mathscr{C}_{/TI}$.

It is a nice exercise to check that this setup indeed gives a strength for the composite monad $T\mathcal{E}$ with respect to $U : \mathscr{C}_{/TI} \to \mathscr{C}$, more precisely:

Theorem 1. *Let $E \in \mathscr{C}$, $\mathcal{E}$ the exception monad $- \oplus E$, and $T\mathcal{E}$ the monad obtained from the distributive law of monads $\mathcal{E}T \to T\mathcal{E}$. Then the monad $T\mathcal{E}$ underlies a strong monad on the $\mathscr{C}^{+}$-category $(\mathscr{C}_{/TI}, *)$, in the terminology of Melliès [36], where $*$ is the pseudo-action defined by composition with $-_1 * -_2 \stackrel{\text{def}}{=} (U -_1) \otimes -_2$.*

Now, unlike usual resource modalities that add duplication and/or erasure (5), $\mathscr{C}_{/TI}$ *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.

Nevertheless, just as linear models with a resource modality "!" give rise to an intuitionistic model by the Girard translations [19], one can build an *ordered CBPV* model whose (positive) objects are those of $\mathscr{C}_{/TI}$, as an instance of a general principle generalising the Girard translation [36, 11] applied to the resource modality 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 further complicates 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 $\mathscr{C}_{/TI}$. 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 without fail.

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. The morphism **new** $: I \to T(R \oplus I)$ is defined by popping an element from the list, or returning the error I if the list is empty.

The morphism $\mathbf{delete} : R \to T\,I$ is defined by pushing the resource onto the list; this is guaranteed to never fail.

Observe that an effectful, closed program of type A in this setup corresponds to a morphism $I \to ([R] \multimap A \otimes [R])$. Its execution consists of supplying an initial list of available resources (*free-list*). 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 free-list 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 free-list 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 free-list is* identical *to the initial free-list*. 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 $B \circ\!\!- A$ 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 $B \circ\!\!- A$ 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 we introduced the *allocation monad* modelling an idealised global "free-list" allocator, with the purpose of studying resource-safety properties. In **Section 2** we define a *linear CBPV* 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. Then, in **Section 3** we define and prove *resource-safety properties* based on the allocation monad.

In **Section 4** we present the *resource CBPV*, an extension of the ordered language, through a calculus $\mathcal{O}_{\mathcal{E},\mathbf{move}}$. It is an affine CBPV language with allocation effects and exceptions, in which every positive type has a chosen effectful erasure map (**drop**), and in which allocation failure results in an exception. Its exchange rule does a side-effect, as it changes the observable execution order of destructors; we also consider its ordered fragment $\mathcal{O}_{\mathcal{E}}$. We define the semantics of $\mathcal{O}_{\mathcal{E},\mathbf{move}}$ by a translation into $\mathcal{L}$ that preserves the ordered fragment.

$$\begin{array}{ccc} \mathcal{O}_{\mathcal{E}} & \xrightarrow{[\![-]\!]} & \mathcal{O} \\ \cap & & \cap \\ \mathcal{O}_{\mathcal{E},\mathbf{move}} & \xrightarrow{[\![-]\!]} & \mathcal{L} \end{array}$$

$\mathcal{O}_{\mathcal{E},\mathbf{move}}$ and $\mathcal{O}_{\mathcal{E}}$ inherit the resource-safety properties of $\mathcal{L}$ and $\mathcal{O}$, respectively.

Then, in **Section 5** we define a presheaf model for $\mathcal{O}$, which models an ordered logic with a destructor for all types, without being a model of affine logic in a traditional sense. Lastly, in **Section 6**, we conclude with a discussion placing the results in the context of the theory of programming languages.

2 An ordered effectful calculus of allocations

In this section, we describe a calculus with allocation effects **new**, **delete**, and with an ordered type system ($\mathcal{O}$) which optionally can be upgraded to linear ($\mathcal{L}$). We provide a common small-step operational semantics modelling this concrete effect. Our starting point is a simplified version of the linear CBPV calculus [11]: for simplicity we consider a calculus given in natural deduction, as opposed to sequent calculus, as we do not investigate the reductions and conversions on open terms. Starting from the linear CBPV calculus, this amounts to considering the derivation of natural deduction rules in the sequent calculus of [11], which directly gives an operational semantics in the form of typed abstract machines in head reduction. It is then adapted to the allocation effect by adding a list of resources, the free-list, to every machine configuration.

Ordered logic refines linear logic by removing the exchange rule, due to which the order of formulae in the antecedent matters for provability. As in Walker [58], we consider a sequencing (composition) rule restricted as follows:

$$\frac{\Gamma, A \vdash B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}\ \text{let}$$

Unlike [48, 58], we consider left functions:

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash B \circ\!\!- A}\ \circ\!\!-_i \qquad\qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B \circ\!\!- A}{\Gamma, \Delta \vdash B}\ \circ\!\!-_e$$

instead of right functions whose rules are symmetric:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ \multimap_i \qquad\qquad \frac{\Gamma \vdash A \multimap B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}\ \multimap_e$$

The notation for left functions might seem unusual but one can think of the notation for an exponential object B^A, and it is justified by the isomorphism:

$$(C \circ\!\!- B) \circ\!\!- A \cong C \circ\!\!- (B \otimes A) \qquad A \multimap (B \multimap C) \cong (A \otimes B) \multimap C$$

We will justify the choice of left vs. right functions in section 3.

CBPV will play an important role for the translation in section 4, and is also interesting in its own right since its treatment of the arrow type is useful to comprehend the multiplicity of closure types in the present of first-class resources (as we will discuss in section 6). Concretely, connectives and types are classified into two kinds, positive and negative, which require different evaluation strategies. Notably, the type R of allocated resources is positive.

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

Expressions of positive type evaluate eagerly, whereas expressions of negative type evaluate lazily. We introduce the notation $\varepsilon ::= + \mid -$ for polarities, and associate to each type its polarity $\varpi(A)$ with $\varpi(P) = +$ and $\varpi(N) = -$.

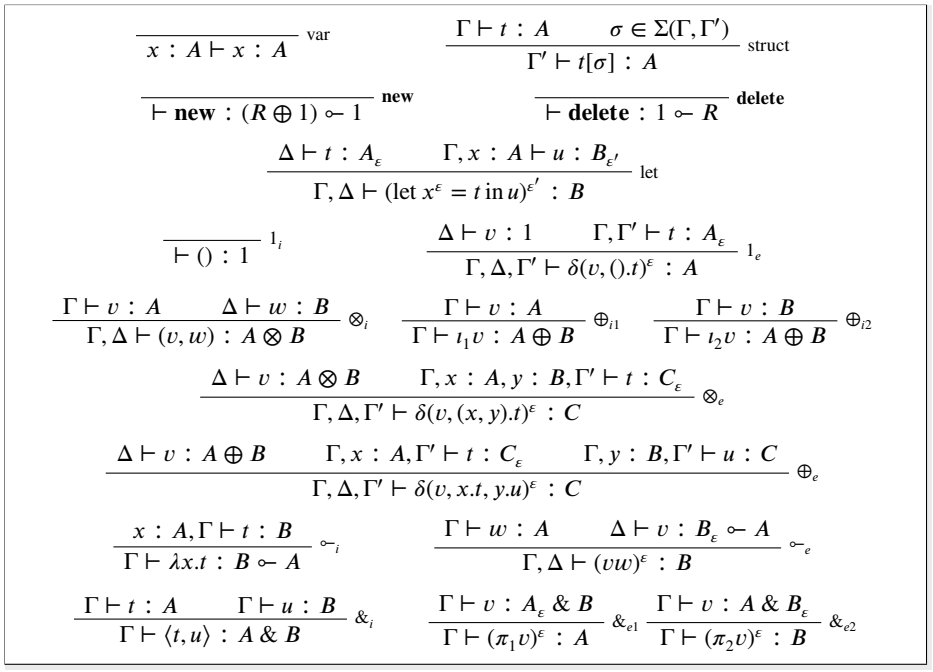

Fig. 1. Typing rules for $\mathcal{L}$

2.1 Grammar of terms

We define *expressions* and *values* with the following grammars:

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

Our untyped expressions do not have typing annotations. Instead, expressions corresponding to elimination rules and let bindings have polarity annotations, which is the minimal amount of information determined by the type that we need to know the 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. Positive values are eliminated with pattern matching: we note these with a dedicated δ eliminators for units, strict pairs and sums. Negative value eliminators follow the usual notation from lambda-calculus: lazy pair projections and function applications.

2.2 Typing rules

We define in fig. 1 typing rules for $\mathcal{L}$. Type polarity annotations in subscript A_ε assert that A has polarity ε. 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. The term $t[\sigma]$ appearing in the *struct*(ural) rule denotes t where all variables have been substituted according to σ; in effect (struct) contains the unrestricted exchange rule. For $\mathcal{O}$ we simply restrict σ to be order-preserving.

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.

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

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

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

2.3 Purely-positive types without resources

An important notion that we will need is that of *purely-positive types without resources* defined as follows:

$$W ::= 1 \mid W \otimes W' \mid W \oplus W' \qquad (13)$$

These types are in particular *central*: they commute for $\otimes$ with any other types, naturally so. We indeed can define by induction values

$$\text{swap}^A_W \;:\; W \otimes A \multimap A \otimes W$$

(corresponding to natural isomorphisms in the semantics of section 5):

$$\begin{aligned}
\text{swap}^A_I &\stackrel{\text{def}}{=} \lambda p.\delta\Big(p,(a,i).\delta\big(i,().(\star,a)\big)\Big)\\
\text{swap}^A_{W_1\otimes W_2} &\stackrel{\text{def}}{=} \lambda p.\delta\Big(p,(a,w).\delta\big(w,(x,x').t_{x,x'}\big)\Big) \text{ where}\\
t_{x,x'} &= \Big(\text{let } p_1 = \text{swap}^{A\otimes W_1}_{W_2}((a,x),x') \text{ in } \delta\big(p_1,(y',p).\delta(p,(a,y).u_{y,y'})\big)\Big)\\
u_{y,y'} &= \Big(\text{let } p_2 = \text{swap}^{W_2\otimes A}_{W_1}((y',a),y) \text{ in } \delta\Big(p_2,(z,p).\delta\big(p,(z',a).((z,z'),a)\big)\Big)\Big)\\
\text{swap}^A_{W_1\oplus W_2} &\stackrel{\text{def}}{=} \lambda p.\delta\big(p,(a,w).\delta(w,x.t_x,y.u_y)\big) \text{ where}\\
t_x &= \Big(\text{let } p_1 = \text{swap}^A_{W_1}(a,x) \text{ in } \delta\big(p_1,(x',a).(\iota_1 x',a)\big)\Big)\\
u_y &= \Big(\text{let } p_2 = \text{swap}^A_{W_2}(a,y) \text{ in } \delta\big(p_2,(y',a).(\iota_2 y',a)\big)\Big)
\end{aligned}$$

$$
\begin{array}{rlrl}
\langle (\text{let } x^- = v \text{ in } t)^\epsilon \,|\, s \,|\, l\rangle^\epsilon & \rightsquigarrow \langle t[v/x] \,|\, s \,|\, l\rangle^\epsilon & & \\
\langle (\text{let } x^+ = t \text{ in } u)^\epsilon \,|\, s \,|\, l\rangle^\epsilon & \rightsquigarrow \langle t \,|\, (x^+.u)^\epsilon \cdot s \,|\, l\rangle^+ & \langle v \,|\, (x^+.t)^\epsilon \cdot s \,|\, l\rangle^+ & \rightsquigarrow \langle t[v/x] \,|\, s \,|\, l\rangle^\epsilon \\
\langle (vw)^\epsilon \,|\, s \,|\, l\rangle^\epsilon & \rightsquigarrow \langle v \,|\, w^\epsilon \cdot s \,|\, l\rangle^- & \langle \lambda x.t \,|\, v^\epsilon \cdot s \,|\, l\rangle^- & \rightsquigarrow \langle t[v/x] \,|\, s \,|\, l\rangle^\epsilon \\
\langle (\pi_i v)^\epsilon \,|\, s \,|\, l\rangle^\epsilon & \rightsquigarrow \langle v \,|\, \pi_i^\epsilon \cdot s \,|\, l\rangle^- & \langle \langle t_1, t_2\rangle \,|\, \pi_i^\epsilon \cdot s \,|\, l\rangle^- & \rightsquigarrow \langle t_i \,|\, s \,|\, l\rangle^\epsilon \\
\langle \delta((v,w),(x,y).t)^\epsilon \,|\, s \,|\, l\rangle^\epsilon & \rightsquigarrow \langle t[v/x, w/y] \,|\, s \,|\, l\rangle^\epsilon & \langle \mathbf{new} \,|\, () \cdot s \,|\, \text{Nil}\rangle^- & \rightsquigarrow \langle \iota_2() \,|\, s \,|\, \text{Nil}\rangle^+ \\
\langle \delta((),().t)^\epsilon \,|\, s \,|\, l\rangle^\epsilon & \rightsquigarrow \langle t \,|\, s \,|\, l\rangle^\epsilon & \langle \mathbf{new} \,|\, () \cdot s \,|\, r_n :: l\rangle^- & \rightsquigarrow \langle \iota_1 r_n \,|\, s \,|\, l\rangle^+ \\
\langle \delta(\iota_i v, x_1.t_1, x_2.t_2)^\epsilon \,|\, s \,|\, l\rangle^\epsilon & \rightsquigarrow \langle t_i[v/x_i] \,|\, s \,|\, l\rangle^\epsilon & \langle \mathbf{delete} \,|\, r_n \cdot s \,|\, l\rangle^- & \rightsquigarrow \langle () \,|\, s \,|\, r_n :: l\rangle^+
\end{array}
$$

Fig. 2. Reduction rules

These types also happen to be discardable ($1 \multimap A$) and copyable ($A \otimes A \multimap A$) but we only rely on centrality in what follows.

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, a free-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 perform the substitution them immediately, following call-by-name reduction, while let bindings of positive expressions push their continuation on the stack to first reduce the expression to a value. All rules that push arguments on the stack come with their dual rule that consumes the argument on the stack. Pattern-matching rules can reduce immediately as only values can be bound. Finally, rules for constants **new** and **delete** are the only ones that manipulate the free-list of resources.

If we interpret our previous program $p = (12)$ with the list of resources $r_0 :: r_1 :: l$, we obtain in particular the following reduction steps:

$$
\begin{aligned}
&\langle p \mid \star \mid r_0 :: r_1 :: l\rangle^+ \\
&\rightsquigarrow^* \left\langle \iota_1 r_1 \;\middle|\; \left(x^+.\delta\big(x, s.(\mathbf{delete}\, s; \mathbf{delete}\, r_0), i.(i; \mathbf{delete}\, r_0)\big)\right) \cdot \star \;\middle|\; l \right\rangle^+ \\
&\rightsquigarrow^* \langle () \mid \star \mid r_0 :: r_1 :: l\rangle^+
\end{aligned}
$$

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

$$\frac{s : A_\epsilon \vdash_p C}{\pi_1^\epsilon \cdot s : A \,\&\, B \vdash_p C} \qquad \frac{s : B_\epsilon \vdash_p C}{\pi_2^\epsilon \cdot s : A \,\&\, B \vdash_p C}$$

Fig. 3. Stack typing rules

2.5 Properties

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

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

We define typing judgements for expressions $\vdash_p t : A$ by extending all typing rules given in section 2.2 with the following axiom scheme for resources: $\vdash_p r_n : R$, to type resources that occur in commands through a program execution. We then extend those judgements in fig. 3 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 3 (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 4 (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 $v_t ::= () \mid (v, w) \mid \iota_i v \mid r_n \mid \langle t, u \rangle \mid \lambda x.t \mid \mathbf{new} \mid \mathbf{delete}$. They include in particular all values v such that $\vdash_p v : A_+$.

Theorem 5 (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[4] 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 property for $\mathcal{L}$ pertaining to its linear character. This property is stated for *complete programs*: closed expressions that return a purely

[4] Although we do not have non-terminating features, our proof by progress & subject reduction does not rely on termination.

positive type without resource, executed in an empty context. We show that for such a program t, the execution starting with any free-list leaves it unchanged upon return, up to a permutation σ:

$$\langle t \mid \star \mid l \rangle \rightsquigarrow^* \langle v \mid \star \mid l' \rangle \qquad \Rightarrow \qquad \exists \sigma, l' = \sigma(l)$$

The property is indeed about resource safety as it provides several good properties expected in programs that manipulate resources:

- all allocated resources are released by the end of the program,
- only previously-allocated resources are released,
- no resource has been released twice (i.e. no "use after free", in this limited context where the only way to use a resource is to release it),
- these properties remain true in case of an error during the program execution, including an allocation error.

We start by describing a stronger property for $\mathcal{O}$: in the ordered language, the final free-list is identical to the initial one ($\sigma = \mathrm{id}$). This corresponds to the expected property with ordered type systems that resources are freed in a LIFO order [58, §1.4]. We also provide a counter-example showing that we cannot include right functions in $\mathcal{O}$ without breaking the LIFO property.

3.1 Ordered case

Let W a purely positive type without resource and $\vdash t : W$ a closed typed expression of $\mathcal{O}$ (i.e. without exchange rule).

Proposition 6. *For any value v and 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 prove this property, we will define the list of resources $LR(t)$ of a term t by tracking resources in contexts of typing rules, and show that such lists are invariant by reduction. In order to reason about resources in case of substitutions, $LR(t[v/x])$ must be definable in terms of $LR(t)$ and $LR(v)$, hence their concatenation. To ensure this (see lemma 8), we restrict 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 $+\!\!+$ the concatenation operation for lists, $\Theta = \Gamma_\Theta; L_\Theta; \Delta_\Theta$ and $\Theta' = \Gamma_{\Theta'}; L_{\Theta'}; \Delta_{\Theta'}$, the expression $\Theta \mathbin{@} \Theta'$ asserts that we are in one of the three following cases to define the concatenation of both contexts:

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

$$\frac{}{;[];x\vdash_o x}\qquad \frac{}{;[];\vdash_o ()}\qquad \frac{}{;[];\vdash_o \mathbf{new}}\qquad \frac{}{;[];\vdash_o \mathbf{delete}}\qquad \frac{}{;[r_n];\vdash_o r_n}$$

$$\frac{\Gamma;[];x,\Delta\vdash_o x}{\Gamma,x;[];\Delta\vdash_o x}\qquad \frac{\Gamma,x;[];\Delta\vdash_o x}{\Gamma;[];x,\Delta\vdash_o x}$$

$$\frac{\Theta\vdash_o v}{\Theta\vdash_o \iota_i v}\qquad \frac{\Theta\vdash_o v}{\Theta\vdash_o \pi_i v}\qquad \frac{\Theta\vdash_o t\qquad \Theta\vdash_o u}{\Theta\vdash_o \langle t,u\rangle}\qquad \frac{x,\Theta\vdash_o t}{\Theta\vdash_o \lambda x.t}$$

$$\frac{\Theta\vdash_o v\qquad \Theta'\vdash_o w}{\Theta\mathbin{@}\Theta'\vdash_o (v,w)}\qquad \frac{\Theta,x\vdash_o u\qquad \Theta'\vdash_o t}{\Theta\mathbin{@}\Theta'\vdash_o \mathrm{let}\ x=t\ \mathrm{in}\ u}$$

$$\frac{\Theta,x,y\mathbin{@}\Theta''\vdash_o t\qquad \Theta'\vdash_o v}{\Theta\mathbin{@}\Theta'\mathbin{@}\Theta''\vdash_o \delta(v,(x,y).t)}\qquad \frac{\Theta\mathbin{@}\Theta''\vdash_o t\qquad \Theta'\vdash_o v}{\Theta\mathbin{@}\Theta'\mathbin{@}\Theta''\vdash_o \delta(v,().t)}$$

$$\frac{\Theta,x\mathbin{@}\Theta''\vdash_o t\qquad \Theta,y\mathbin{@}\Theta''\vdash_o u\qquad \Theta'\vdash_o v}{\Theta\mathbin{@}\Theta'\mathbin{@}\Theta''\vdash_o \delta(v,x.t,y.u)}\qquad \frac{\Theta\vdash_o w\qquad \Theta'\vdash_o v}{\Theta\mathbin{@}\Theta'\vdash_o vw}$$

Fig. 4. Typing rules of ordered expressions with resources (types omitted)

$$\frac{}{[]\vdash_o^S \star}\qquad \frac{;L_v;\vdash_o v\qquad L_s\vdash_o^S s}{L_s\mathbin{+\!\!+}L_v\vdash_o^S v\cdot s}$$

$$\frac{;L_t;x\vdash_o t\qquad L_s\vdash_o^S s}{L_s\mathbin{+\!\!+}L_t\vdash_o^S (x^+.t)\cdot s}\qquad \frac{;L_t;\vdash_o t\qquad L_s\vdash_o^S s}{L_s\mathbin{+\!\!+}L_t\mathbin{+\!\!+}l\vdash_o^C \langle t\mid s\mid l\rangle}$$

Fig. 5. Typing rules of ordered stacks and commands with resources (types omitted)

We can now type terms with resources with the judgement $\vdash_o$ indexed by such contexts, which enrich the previous typing judgements with information to track their resources (in fig. 4, where types are omitted 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 7. *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.

Lemma 8. (Left SL) *If $;L_t;\vdash_o t$ and $\Gamma,x;L_u;\Delta\vdash_o u$, then $\Gamma;L_t\mathbin{+\!\!+}L_u;\Delta\vdash_o u[t/x]$.* (Right SL) *If $;L_t;\vdash_o t$ and $\Gamma;L_u;x,\Delta\vdash_o u$, then $\Gamma;L_u\mathbin{+\!\!+}L_t;\Delta\vdash_o u[t/x]$ (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 9. *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 6:

Proof (proposition 6). For any expression with a purely-positive type without resource $\vdash t : W$ 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 resources. Since reduction rules preserve the list of resources, we have $l = l'$.

3.2 A counter-example for right functions

We only have left functions in $\mathcal{O}$, that is, abstractions binding the leftmost variable together with the matching elimination rule from ordered logic. We provide the following counter-example to the resource-safety property with right functions. We underline the right function abstractions and applications.

$$p \stackrel{\text{def}}{\equiv} \delta^*\Big(\mathbf{new}(), r.\delta^*\big(\mathbf{new}(), s.(\underline{t_r s}), i.(\mathbf{delete}\, r; i)\big), i.i\Big)$$
$$\text{where } t_r = \big(\text{let } i = \mathbf{delete}\, r \text{ in } \delta(i, ().\underline{\lambda x}.\, \mathbf{delete}\, x)\big)^- \tag{14}$$

This example does not belong to $\mathcal{O}$ as it uses the right function $t_r : R \multimap 1$ and its application $\underline{t_r s}$.

$$\dfrac{\dfrac{\dfrac{}{r : R \vdash \mathbf{delete}\, r : 1}\,{\scriptstyle \multimap_e} \quad \times \dfrac{\dfrac{\dfrac{}{x : R \vdash \mathbf{delete}\, x : 1}\,{\scriptstyle \multimap_e}}{\vdash \underline{\lambda x}.\, \mathbf{delete}\, x : R \multimap 1}\,{\scriptstyle \multimap_i \text{ (not in } \mathcal{O})} \quad \dfrac{}{i : 1 \vdash \delta(i, ().\underline{\lambda x}.\, \mathbf{delete}\, x) : R \multimap 1}\,{\scriptstyle 1_e}}{r : R \vdash \text{let } i = \mathbf{delete}\, r \text{ in } \delta(i, ().\underline{\lambda x}.\, \mathbf{delete}\, x) : R \multimap 1}\,\text{let}}{\times \dfrac{r : R \vdash t_r : R \multimap 1 \quad = \quad \dfrac{}{s : R \vdash s : R}}{r : R, s : R \vdash \underline{t_r s} : 1}\,{\scriptstyle \multimap_e \text{ (not in } \mathcal{O})}}$$

As we can see from the typing derivation, the context is ordered as r, s, which is the converse of what we would have obtained with a left function. Subsequently, we can then calculate that two resources r_0 and r_1 are reordered during the execution as follows (with the obvious reductions rules for the right abstraction and application):

$$\langle p \mid \star \mid r_0 :: r_1 :: l\rangle \rightsquigarrow^* \langle () \mid \star \mid r_1 :: r_0 :: l\rangle$$

This contradicts the LIFO property for $\mathcal{O}$.

3.3 Non-ordered case

If we consider expressions $\vdash t : W$ in $\mathcal{L}$, that is by adding the exchange rule, they have a weaker property:

Theorem 10. *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 $l' = \sigma(l)$.*

This is proved similarly to proposition 6, 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 The resource call-by-push-value

We now introduce a resource CBPV: an ordered variant of CBPV with an allocation effect **new**, exceptions, a chosen effectful destructor **drop** at all types, and a **move** operation that implements a side-effecting exchange rule.

Formally, we define a calculus $\mathcal{O}_{\mathcal{E},\mathbf{move}}$ which extends $\mathcal{O}$. Its semantics is given by translation into $\mathcal{L}$: 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 gives resource-safety properties for $\mathcal{O}_{\mathcal{E},\mathbf{move}}$. In addition, by removing **move**, we obtain a calculus $\mathcal{O}_{\mathcal{E}}$ whose translation falls into $\mathcal{O}$.

As a starting point, we picture the situation described in section 1.4:

$$\mathscr{C}_{/TI} \underset{F^{\mathcal{D}}}{\overset{U^{\mathcal{D}}}{\rightleftarrows}} \mathscr{C} \underset{U^{T\mathcal{E}}}{\overset{F^{T\mathcal{E}}}{\rightleftarrows}} \mathscr{C}^{T\mathcal{E}} \qquad \bot\,(a),\ \bot\,(b) \tag{15}$$

where $\mathcal{D} = - \,\&\, TI$ provides, for an arbitrary linear type, the free type with destructor (given by its second projection).

When trying to add exceptions to a general notion of effect given by a strong monad T, one would look at the adjunction (b), which, assuming $T\mathcal{E}$ strong, gives rise to a linear CBPV model [11], with positive types interpreted in $\mathscr{C}$ and negative types interpreted in $\mathscr{C}^{T\mathcal{E}}$. It is unnatural, though, to assume that $T\mathcal{E}$ is strong. In order to recover a strength for $T\mathcal{E}$, we want to restrict the positive types to the linear types that are provided with a chosen destructor, by looking at the monoidal adjunction (a) above. When starting from an adjunction model, composing with a monoidal adjunction on the left yields another adjunction model, a construction which generalises the Girard translations [11, §5.3] (see also [42] for more details). This suggests that we look at the adjoint situation

$$\mathscr{C}_{/TI} \underset{\downarrow}{\overset{\uparrow}{\rightleftarrows}} \mathscr{C}^{T\mathcal{E}} \qquad \bot$$

obtained by composition with $\uparrow \stackrel{\text{def}}{=} F^{T\mathcal{E}}U^{\mathcal{D}}$ and $\downarrow \stackrel{\text{def}}{=} F^{\mathcal{D}}U^{T\mathcal{E}}$. Note that in terms of underlying types, one has $\uparrow A = T(A \oplus E)$ and $\downarrow A = A \,\&\, TI$.

We cannot apply the results of [11] directly, which only deals with the situation where the monoidal categories are symmetric and the monad is strong. Instead, we do this construction by hand, which we give as a translation of the resource CBPV into $\mathcal{O}$ and $\mathcal{L}$. Within this more focused approach, we refine the translation underlying [11, §5.3]

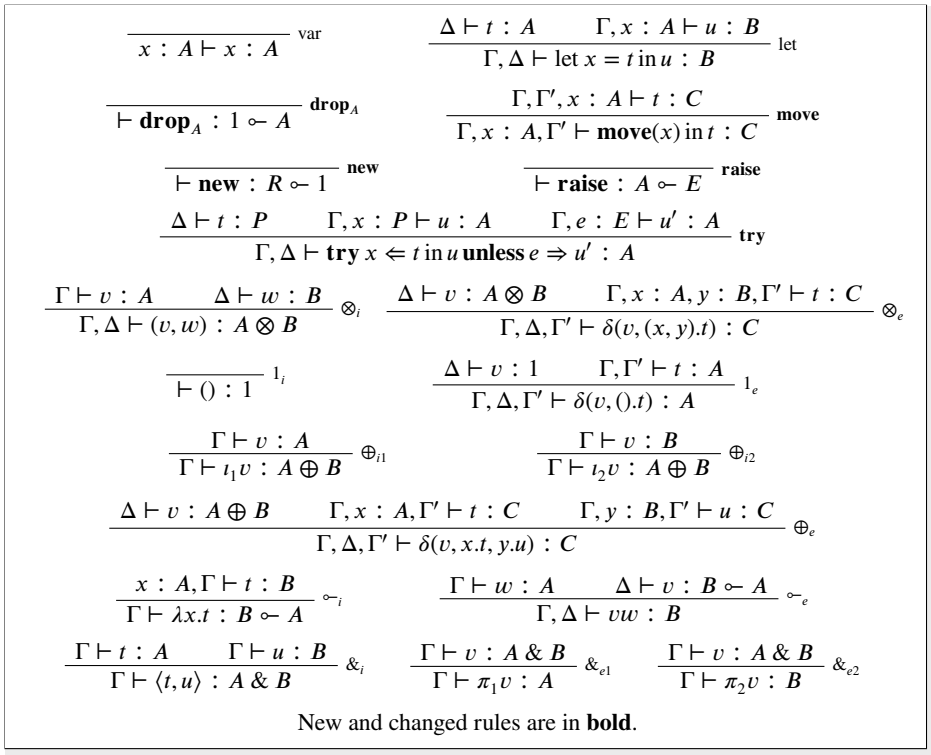

Fig. 6. Typing rules for $\mathcal{O}_{\mathcal{E},\mathbf{move}}$

with ordered sequents, and with the consideration of a strength of $T\mathcal{E}$ with respect to U^D, which is necessary for having antecedents with several variables in the first place.

Another important difference is that we are translating direct-style calculi into direct-style calculi: the monad T is, already, the ambient monad for side-effects in $\mathcal{O}$ and $\mathcal{L}$. As it turns out, we can still adapt the translation to work in this way. Note that the type $\downarrow A = A \,\&\, TI$ is $A \,\&\, 1$ in $\mathcal{O}$, and that the type $\uparrow A = T(A \oplus E)$ describes effectful computations of type $A \oplus E$ in $\mathcal{O}$. In what follows, we make use of these type constructions in $\mathcal{O}$ which we note $\Downarrow A \stackrel{\text{def}}{=} A \,\&\, 1$ and $\Uparrow A \stackrel{\text{def}}{=} A \oplus E$.

4.1 Expressions and types

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

- "**new**", now of type $1 \multimap R$, which raises an exception in case of allocation error.
- "$\mathbf{drop}_A$", a function of type $1 \multimap A$ that releases the resources contained in its argument.
- "$\mathbf{move}(x)$ in t", an expression that places x on top of the stack in t.
- "**raise**", a value that inhabits any type with a given exception.
- "$\mathbf{try}\ x \Leftarrow t \text{ in } u \ \mathbf{unless}\ e \Rightarrow u'$", an expression that catches exceptions occuring in t based on [4].

We call $\mathcal{O}_{\mathcal{E}}$ the fragment of $\mathcal{O}_{\mathcal{E},\mathbf{move}}$ without **move**. Also, from now on we leave polarity annotations implicit because they can always be inferred from the type.

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, since each positive type is assigned a chosen destructor.

The calculus is parameterised by a type of exceptions E. In $\mathcal{O}_{\mathcal{E}}$, exceptions must not exchange resources during stack unwinding, so we add the constraint that E *is a purely-positive type without resource*. We also assume given some closed value $\vdash \mathrm{Alloc\_failure} : E$, which is the exception raised whenever an allocation fails. (For instance, $E = 1$ and $\mathrm{Alloc\_failure} = ()$.) We will rely on terms swap^{A}_{W} in $\mathcal{O}_{\mathcal{E}}$ defined as in $\mathcal{O}$ (section 2.3).

4.2 Move as an effectful exchange rule

As explained in section 1.4, the category $\mathscr{C}_{/TI}$ in the situation (15) depicted above is not symmetric in general. However, when $\mathscr{C}$ is symmetric, there is nevertheless an effectful map $A \otimes B \to B \otimes A$, as found in the following hom-set:

$$\mathscr{C}_{/TI}(A \otimes B, F^{D}U^{D}(B \otimes A)) \cong \mathscr{C}(U^{D}A \otimes U^{D}B, U^{D}B \otimes U^{D}A)$$

This is reflected in $\mathcal{O}_{\mathcal{E},\mathbf{move}}$ with the operation **move** which exchanges variables. It is not presented as a structural rule in the traditional way: it indeed performs an effect and does not commute with other rules. In the following example, two terms allocate three resources and then frees them in reverse order. They would be identified if **move** was treated as a structural rule in the usual manner (e.g. as in $\mathcal{L}$), but they behave differently:

$\mathrm{let}\ r = \mathbf{new}()\ \mathrm{in}$	$\mathrm{let}\ r = \mathbf{new}()\ \mathrm{in}$
$\mathrm{let}\ s = \mathbf{new}()\ \mathrm{in}$	$\mathrm{let}\ s = \mathbf{new}()\ \mathrm{in}\ \mathbf{move}(r)\ \mathrm{in}$
$\mathrm{let}\ t = \mathbf{new}()\ \mathrm{in}$	$\mathrm{let}\ t = \mathbf{new}()\ \mathrm{in}$
$\mathbf{drop}_R\, t; \mathbf{drop}_R\, s; \mathbf{drop}_R\, r$	$\mathbf{drop}_R\, t; \mathbf{move}(s)\ \mathrm{in}\ \mathbf{drop}_R\, s; \mathbf{drop}_R\, r$

Both terms allocate three resources then frees them in reverse order. If the allocation of t fails, then the raised exception will free the first two resources in a different order, which can be observed with the evaluation context $\langle \star \mid [r_0, r_1] \rangle$: the final free-lists are respectively $[r_0, r_1]$ and $[r_1, r_0]$.

For this reason, **move** cannot give rise to a *contextual isomorphism* [32] between $A \otimes B$ and $B \otimes A$ for arbitrary A and B, that is, which would allow us to replace $A \otimes B$ with $B \otimes A$ in arbitrary context without changing the meaning. Since a contextual isomorphism just requires inverses and purity (thunkability) [32], **move** must be effectful in any reasonable interpretation.

4.3 Translation into $\mathcal{L}$

Each type A has a positive interpretation A^+ equipped with a destructor written drop_A and a negative interpretation A^-.

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

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

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

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

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

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

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

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

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

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

We add explicit coercions to treat positive values as expressions with the notation $\mathrm{coerc}(v)$. 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.

$$
ax : [\![x \vdash x : A]\!] \stackrel{\text{def}}{=} x
$$

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

Note that expressions $t : P$ have a second component that is immediately discarded, and they are evaluated eagerly. Hence, they are essentially expressions of type $A \oplus E$. The operational semantics of an affine value v is that of $[\![v]\!]$, and the operational semantics of an affine expression t is that of $\pi_1 [\![t]\!]$.

Proposition 11. *The translation defined above is well-defined: it preserves typing, and it uses the exchange rule in the target only in the translation of* **move***, so the translation of the $\mathcal{O}_{\mathcal{E}}$ fragment indeed falls into $\mathcal{O}$.*

More details for the preservation of typing are given in section D.1.

Theorem 12. *For any expression t of $\mathcal{O}_{\mathcal{E},\mathbf{move}}$ with a purely-positive type without resource W, if $\langle \pi_1 [\![t]\!] \mid \star \mid l \rangle \rightsquigarrow^* \langle v \mid \star \mid l' \rangle$ then $l' = \sigma(l)$ for some permutation σ. Moreover, if the expression does not contain "***move***", then $l' = l$.*

Proof. Since purely-positive types without resources are translated into themselves, one has $\Downarrow W^- = (W \oplus E) \,\&\, I$, hence $\pi_1 [\![t]\!]$ is a linear expression of type $W \oplus E$. By hypothesis E is purely-positive without resource, and so $W \oplus E$ is too. Hence, the resource-safety property of $\mathcal{L}$ applies to $\pi_1 [\![t]\!]$, as well as that of $\mathcal{O}$ if t does not contain "**move**". Those are precisely the properties we needed to prove.

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 $\mathscr{C} \stackrel{\text{def}}{=} Set^{[\mathbf{R}]}$, 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 [12] lifts it to a non-symmetric monoidal product in $\mathscr{C}$ with a right closed and left closed structure, corresponding to the multiplicative fragment. We define $R \in Set^{[\mathbf{R}]}$ to be the indicator function for singleton lists. As a presheaf category, $\mathscr{C}$ has products and coproducts. The initial algebra of $1 \oplus (R \otimes -)$ interpreting lists of resources is the terminal object $\top$. Concretely, we have the following constructions in $\mathscr{C}$:

- The monoidal product $(A \otimes B)(l) \stackrel{\text{def}}{=} \sum_{(l_1,l_2)\in[R]^2 \wedge (l_1 +\!\!+ l_2 = l)} A(l_1) \times B(l_2)$.
- The unit $1([]) \stackrel{\text{def}}{=} \{()\}, 1(l) \stackrel{\text{def}}{=} \emptyset$ is the indicator function for the empty list.
- The right arrow $(A \multimap B)(l_1) \stackrel{\text{def}}{=} \prod_{l_2\in[R]} A(l_2) \to B(l_1 +\!\!+ l_2)$.
- The left arrow $(B \multimap A)(l_2) \stackrel{\text{def}}{=} \prod_{l_1\in[R]} A(l_1) \to B(l_1 +\!\!+ l_2)$.
- 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 exploit the fact that the canonical adjunction

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

for the allocation monad is made of endofunctors such that

$$A \otimes FB \cong F(A \otimes B) \tag{16}$$

to interpret positive and negative terms in the same category: this allows us to adapt the model theory of [11] while avoiding the formalism of enriched categories.

We now have all the ingredients to interpret derivations of $\mathcal{O}$ (fig. 1). In fact, we will interpret derivations of ordered terms with resources from fig. 4 which extends the typing rules of $\mathcal{O}$. Expressions typed with $\Gamma; [r_1 \cdots r_n]; \Delta \vdash_o A$ are interpreted as (oblique) morphisms in $\mathscr{C}(F(\Gamma \otimes R^n \otimes \Delta), A)$, following [11] which we adapt to the ordered case (see appendix E for the complete definition of the interpretation).

Interpreting the left function and the right let binding uses the isomorphism (16) as a left strength. We cannot interpret a right function or left let-binding for $\mathcal{O}$, as it would require an isomorphism $FA \otimes B \cong F(A \otimes B)$ which swaps the ambient list of resources with B. This corresponds to the re-ordering of resources by the incorrect term t_r (14).

Lemma 13 (Substitution lemma). *For all $\Theta'' \vdash_o v : A$ and $\Theta, x : A \mathbin{@} \Theta' \vdash_o t : B$, one has $[\![t[v/x]]\!] = (id_{\Theta^+} \otimes [\![v]\!] \otimes id_{\Theta'^+}); [\![t]\!]$ (see section E.2).*

Theorem 14 (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 $[\![c]\!] = [\![c']\!]$ (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. We ask for objects that commute with all other objects A in a coherent way. Those are the objects in the *Drinfeld centre* of $\mathscr{C}$ (as defined e.g. in [41, §3]).

Proposition 15. *Given an object E, the three following properties are equivalent:*

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

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

Proposition 16. *Every object in $\mathscr{C}$ has a (unique) destructor, but $\mathscr{C}$ is not a model of affine logic in the sense of having an isomorphism $I \cong \top$ (cf. [9, §3.2]).*

Proof. Recall that $[R] = \top$ is the terminal object. Destructors for A are objects in $(A, \delta) \in \mathscr{C}/TI$; i.e. $\delta \in \mathscr{C}(A, TI) \cong \mathscr{C}(A, [R] \multimap [R]) \cong \mathscr{C}(A \otimes \top, \top)$ 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 $\mathscr{C}(A, I)$, which is not the case for $[R] = \top$.

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

Proposition 17. *Let E a central object. The monad $T\mathcal{E}$ where $\mathcal{E} = (- \oplus E)$ has a left strength $\Gamma \otimes T\mathcal{E}A \to T\mathcal{E}(\Gamma \otimes A)$.*

Thus, this gives a model of an affine logic in the sense of the calculus with destructors $\boldsymbol{\mathcal{O}}_{\mathcal{E}}$, but we do not have a model of affine logic in the usual sense of semi-cartesian monoidal categories. Even if we consider that the monoidal unit I is indeed terminal in the Kleisli category of T, the destructors are not thunkable for T, a common requirement for morphisms interpreting structural rules in premonoidal categories (see e.g. Führmann [17]). This is another way of observing the fact that weakening performs a side effect.

We might wonder how we could extend this model to ones of $\boldsymbol{\mathcal{L}}$ and $\boldsymbol{\mathcal{O}}_{\mathcal{E},\mathbf{move}}$. This amounts to somehow regaining the symmetry of the monoidal product to interpret the exchange rule. One way to do that would be to quotient our base category $[\mathbf{R}]$ to obtain multisets. However, T would become commutative and so $\mathscr{C}_{/TI}$ would be symmetric; intuitively the model does not observe the order of destruction. Instead, adding permutations as isomorphisms in $[\mathbf{R}]$ would result in a presheaf model on a setoid in which T is not commutative; intuitively the model tracks in the interpretation of terms their action on the free-list of resources. As we cannot re-use our trick with endofunctors to avoid the presheaf-enriched formalism, we leave this as future work.

6 Discussion and perspectives

After this formal development, we find it useful to place 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: for instance, every allocated resource in $\mathcal{L}$ and $\mathcal{O}_{\mathcal{E},\mathbf{move}}$ is freed exactly once, even in case of errors during the execution of 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 [47, 48, 58, 45]. 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 the exchange rule with a side-effecting "move" operation 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 (among other) on the problematic interaction of linearity with error handling, another important aspect with resources that are acquired in a program. For instance, Cyclone, a source of inspiration for the Rust language, permitted resource leaks in the name of *"flexibility and usability"* [53, §3.5] (necessitating back-up collection mechanisms). The practical experiment of Tov and Pucella [57, 55], another milestone in exploring practical aspects of linear type systems, also mentioned the issue of combining linearity and control effects, which became the motivation for their *"practical"* linear type system to be affine.

Tov and Pucella [56], separately, seeking to lift the limitation of the affine system for manipulating resources as first-class values, proposed a type system that offers both linearity and control effects (e.g. checked exceptions), by typing effects in addition to linearity. Essentially, effects are constrained when linear variables are in scope (or conversely), matching the restrictive approach we described in section 1.2 corresponding to the notion of strength with respect to an affine or an unrestricted resource modality (8). It should be noted that this system [56], unlike the previous experiment [57], is mainly justified by its mathematical properties; how useful such a system is, in practice, remains unclear.[5]

Lastly, *affine session types* [40] do mix linearity and cancellation in a sense similar to ours. They guarantee a safety property (absence of deadlock) after an exception (cancellation) arises, by communicating cancellation across channels.

Our analysis from the introduction applies equally to explicit error handling with an error monad, as illustrated with the language $\mathcal{L}$. Indeed, manual error handling encounters similar obstacles arising from the lack of unrestricted monadic strength for the error monad. Concretely, for each location where one would like to use a monadic binding (4),

[5] As Tov themselves explains: *"One question that remains, however, concerns the pragmatics of checked exceptions in a higher-order language such as Alms, where latent exception effects are likely to appear on many function arrows. Weighing the cost against the benefit, I have decided that adding linear types to Alms is not worth the complexity of a programmer-visible effect system."* [55, p. 238]

one needs to implement by hand a repetitive description of whichever clean-up functions should be called in case of error propagation—a well-known and tedious problem in C, which gave rise to specific programming patterns involving `goto` (see appendix A).

By 2012, these limitations of linear types with respect to the handling of errors and exceptions, arising from practical considerations, might seem well-established in the literature. However, more recent works on linear type systems purporting to implement practical first-class resources in academic programming languages [6, 44] do not explore nor mention these obstacles. In the case of [6], a section describing limitations arising from Haskell exceptions indeed appears in an earlier version with a different title [7], but is absent from the published peer-reviewed version.

With the resource CBPV, we set out to model some aspects of C++ resources with destructors and move operations. Compared to the works cited previously, C++ realised, much earlier [52], a shift in viewpoint whereby clean-up functions are deduced from the types, whilst it also later pioneered "move semantics" [24] for treating resources as first-class values. As we have seen, interpreting resource types in the slice category over some TI, i.e. as types provided with some chosen effectful weakening map $A \to TI$, 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 as a side-effect. One surprising aspect of the resource CBPV is that it combines facets of all three of affine, linear and ordered logics:

affine	in terms of types and available control effects
linear	in terms of first-class values and resource-safety properties
ordered	in terms of proof-relevant semantics and 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 the lineage of [57]) in order to integrate first-class resources in functional programming languages [43].

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

The resource CBPV is a very idealised model of C++/Rust-style resource management situated at the intersection of linear logic and the theory of effects. Various models of ownership in Rust—more practical and less idealised—have been developed to prove safety and functional properties of Rust programs, such as with separation logic [27], a dedicated functional logic [13], or by translation into a purely functional language [25]. 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, our model is the first 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 an exchange rule in logic, albeit performing a side-effect) is responsible for altering the order of

destruction; and indeed closures themselves being resources. In fact, in the presence of both resource modalities ! and $\mathcal{D}$, different positive types of closures, derived from $\multimap$, coexist ($!(A \multimap B)$ and $\mathcal{D}(A \multimap B)$), distinguished by the restrictions on the typing context in accordance with the resource modality (e.g. copyable or resource-like)—a distinction between several types of closures that can also be seen in C++ and Rust.

One very interesting aspect of the resource CBPV 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:

$$A \otimes B \rightleftarrows B \otimes A \tag{17}$$

without the two types being isomorphic. This is because a *contextual* notion of isomorphism (allowing to substitute equals by equals) also requires the two inverse maps (17) to be pure [32]. This radically departs from symmetric premonoidal categories [49].

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 formulae 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` [18]. 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 [28, 52, 24], and with the use of the slice category in this context being prompted by empirically-observed phenomena with C++ resources [10].

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. Indeed, the functional correctness of some Rust programs can depend on the resource-safety properties enforced by the language, for instance when they use the typestate programming pattern or when they are made fault-tolerant.

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 CBPV [11] which the present work is based on. Though not essential to the results of this paper, it is now clear that extending the Curry-Howard-Lambek correspondence to ordered CBPV would be useful as a meta-theory to study resources. Removing the symmetry in monoidal categories and the exchange rule in logic can create technical issues—we have made some progress 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. A general mechanism of polarities should provide foundations for kind-based linear type systems [57] and extend them to computationally-relevant kinds in the style of Rust's special traits (e.g. `Copy`, `Drop`). Interesting questions arise from the practical necessity of kind polymorphism in such languages.
- **Borrowing** We have modelled resources that can only be allocated and deallocated, which lets us 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 [20, 16], inspired by regions [54], 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 [27, 13, 25], 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.

Acknowledgements

We thank the anonymous reviewers for their detailed and thoughtful comments on earlier versions of this article.

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. Bernardy, J., Boespflug, M., Newton, R.R., Peyton Jones, S., Spiwack, A.: Retrofitting Linear Types (In submission), (2017). https://www.microsoft.com/en-us/research/wp-content/uploads/2017/03/haskell-linear-submitted.pdf.

8. Blute, R., Cockett, J., Seely, R.: ! and ?-Storage as tensorial strength. Mathematical Structures in Computer Science **6**(4), 313–351 (1996)
9. Braüner, T.: A model of intuitionistic affine logic from stable domain theory. In: International Colloquium on Automata, Languages, and Programming, pp. 340–351 (1994)
10. Combette, G., Munch-Maccagnoni, G.: A resource modality for RAII (abstract). In: LOLA 2018: Workshop on Syntax and Semantics of Low-Level Languages (2018). https://hal.inria.fr/hal-01806634
11. 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
12. 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)
13. 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
14. Filinski, A.: Linear Continuations. In: Proc. POPL, pp. 27–38 (1992)
15. Filinski, A.: Representing Monads. In: Proc. POPL, pp. 446–457. ACM Press (1994)
16. 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
17. Führmann, C.: Direct Models for the Computational Lambda Calculus. Electr. Notes Theor. Comput. Sci. **20**, 245–292 (1999)
18. 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
19. Girard, J.-Y.: Linear Logic. Theoretical Computer Science **50**, 1–102 (1987)
20. 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
21. Hasegawa, M.: Linear Exponential Comonads without Symmetry. In: Fourth International Workshop on Linearity (2016). http://arxiv.org/abs/1701.04919
22. 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
23. Hasegawa, M.: Semantics of linear continuation-passing in call-by-name. In: International Symposium on Functional and Logic Programming, pp. 229–243 (2004)
24. 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
25. Ho, S., Protzenko, J.: Aeneas: Rust Verification by Functional Translation. Proc. ACM Program. Lang. **6**(ICFP) (2022). https://doi.org/10.1145/3547647

26. Hofmann, M.: A Type System for Bounded Space and Functional In-Place Update. Nord. J. Comput. **7**(4), 258–289 (2000)
27. 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
28. Koenig, A., Stroustrup, B.: Exception Handling for C++. In: Proceedings of the C++ Conference. San Francisco, CA, USA, April 1990, pp. 149–176 (1990)
29. Lafont, Y.: The linear abstract machine. Theoretical computer science **59**(1-2), 157–180 (1988)
30. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2). Kluwer Academic Publishers, USA (2004)
31. Levy, P.B.: Call-by-Push-Value: A Subsuming Paradigm. In: Proc. TLCA '99, pp. 228–242 (1999)
32. Levy, P.B.: Contextual isomorphisms. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 400–414 (2017)
33. 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)
34. Melliès, P.-A.: "Categorical models of linear logic revisited". working paper or preprint.
35. Melliès, P.-A.: "Categorical semantics of linear logic". In: vol. 27. Panoramas et Synthèses. Société Mathématique de France, 2009. Chap. 1, pp. 15–215.
36. Melliès, P.-A.: "Parametric monads and enriched adjunctions". Draft.
37. 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
38. 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)
39. 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
40. 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. arXiv: 1809.02781v2 [cs.LO]
41. Müger, M.: From subfactors to categories and topology II: The quantum double of tensor categories and subfactors. Journal of Pure and Applied Algebra **180**(1), 159–219 (2003). https://doi.org/10.1016/S0022-4049(02)00248-7
42. Munch-Maccagnoni, G.: Note on models of polarised intuitionistic logic. Tech. rep., INRIA (2017). https://hal.inria.fr/hal-01540760
43. Munch-Maccagnoni, G.: Resource Polymorphism, (2018). https://doi.org/10.48550/arXiv.1803.02796.
44. 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
45. 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
46. 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.
47. Polakow, J.: Ordered Linear Logic and Applications. PhD thesis, Carnegie Mellon University (2001).
48. 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
49. Power, A.J., Robinson, E.: Premonoidal categories and notions of computation. Mathematical Structures in Computer Science **5**(7), 453–468 (1997)
50. 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
51. 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)
52. 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
53. 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)
54. Tofte, M., Talpin, J.-P.: Region-based memory management. Information and computation **132**(2), 109–176 (1997)
55. Tov, J.A.: Practical Programming with Substructural Types. PhD thesis, Northeastern University (2012).
56. 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
57. 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
58. 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 complicates 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/or 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.

We focused in this paper on using a notion of *ownership*, generally understood to be related to linearity, to lift the first limitation using. We elaborated more precisely a formal link between a notion of ownership present in the C++ and Rust programming languages, that has not seen much theoretical study, and linearity. We did not attempt to model the second limitation, which is already the subject of much study involving other theoretical tools.

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. Pseudo-code for manual conditional transfer of ownership

To implement the transfer of ownership, C++ initially experimented with smart pointers such as the now-deprecated `auto_ptr`. So-called "move semantics" was introduced in C++11 to fix the defects of `auto_ptr` in representing ownership and the excessive amounts of deep copies encouraged by the language. So-called "move constructors" and "move assignment operators", associated to a notion of "rvalue reference", enabled the definition of "unique pointers" and a "move" operation to transfer ownership of values, as illustrated 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. C++ pseudo-code using a file to record operations

In effect, `std::move` is responsible for selecting the move constructor of `File` for its argument; the move constructor is then responsible for implementing the move operation which consists (most of the time) in a shallow copy into the target, followed by the nullification of the source one (or at least its replacement with some valid, default value). In this way, it is guaranteed that a single copy of the resource exists at any time, and the destructor of `f` is not called (or more precisely, is called on a null or default value, which does nothing). This notion of ownership in C++ can be described as ensuring that the owner (responible for calling the destructor) is unique.

In order to implement first-class resources, C++11 introduced language and standard library features making it convenient to define resource datatypes, such as the rules for the automatic derivation of move constructors (which, together with similar rules for destructors, are responsible for instance for a struct of resources to behave as a resource in the obvious manner). In actual usage (see e.g. the "rule of zero"), these are very close to defining separate kinds of movable vs. copyable types.

B Details: basic properties

B.1 Proof of swap and definition of swap inverse

We recall the expressions swap^A_W:

$$\mathrm{swap}^A_I \stackrel{\text{def}}{\equiv} \lambda p.\delta(p,(a,i).\delta(i,().(\star,a)))$$

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

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

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

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

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

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

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:

- $\mathrm{swap}^{A}_{I} \stackrel{\text{def}}{=} AI \to A \to IA$.
- $\mathrm{swap}^{A}_{W_1\oplus W_2} \stackrel{\text{def}}{=} A(W_1 \oplus W_2) \to AW_i \to W_iA \to (W_1 \oplus W_2)A$.
- $\mathrm{swap}^{A}_{W_1W_2} \stackrel{\text{def}}{=} A(W_1W_2) \to (AW_1)W_2 \to W_2(AW_1) \to (W_2A)W_1 \to W_1(W_2A) \to (W_1W_2)A$.
- $\mathrm{swap}^{I}_{A} \stackrel{\text{def}}{=} IA \to A \to AI$.
- $\mathrm{swap}^{W_1\oplus W_2}_{A} \stackrel{\text{def}}{=} (W_1 \oplus W_2)A \to W_iA \to AW_i \to A(W_1 \oplus W_2)$.
- $\mathrm{swap}^{W_1W_2}_{A} \stackrel{\text{def}}{=} (W_1W_2)A \to W_1(W_2A) \to (W_2A)W_1 \to W_2(AW_1) \to (AW_1)W_2 \to A(W_1W_2)$.

B.2 Proof of determinism (proposition 2)

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 4)

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

- $\langle (\text{let } x^+ = t \text{ in } u)^\varepsilon \mid s \mid l \rangle^\varepsilon \rightsquigarrow \langle t \mid (x^+.u)^\varepsilon \cdot s \mid l \rangle^+$: by the hypothesis $\langle (\text{let } x^+ = t \text{ in } u)^\varepsilon \mid s \mid l \rangle^\varepsilon : A$, we get $\vdash_p t : B_+, x : B \vdash_p u : C_\varepsilon$ and $s : C \vdash_p A$. Hence, we have $(x^+.u)^\varepsilon \cdot s : B \vdash_p A$ and so $\langle t \mid (x^+.u)^\varepsilon \cdot s \mid l \rangle^+ : A$.
- $\langle (\text{let } x^- = v \text{ in } t)^\varepsilon \mid s \mid l \rangle^\varepsilon \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle^\varepsilon$: by hypothesis, we get $\vdash_p v : B_-, x : B \vdash_p t : C_\varepsilon$ and $s : C \vdash_p A$. By SL, we have $\vdash_p t[v/x] : C$ and so $\langle t[v/x] \mid s \mid l \rangle^\varepsilon : A$.
- $\langle (vw)^\varepsilon \mid s \mid l \rangle^\varepsilon \rightsquigarrow \langle v \mid w^\varepsilon \cdot s \mid l \rangle^-$: by hypothesis, we get $\vdash_p v : C_\varepsilon \multimap B, \vdash_p w : B$ and $s : C \vdash_p A$. Hence, we have $w^\varepsilon \cdot s : C \multimap B \vdash_p A$ and so $\langle v \mid w^\varepsilon \cdot s \mid l \rangle^- : A$.
- $\langle (\pi_i v)^\varepsilon \mid s \mid l \rangle^\varepsilon \rightsquigarrow \langle v \mid \pi_i^\varepsilon \cdot s \mid l \rangle^-$: by hypothesis, we get $\vdash_p v : B_1 \,\&\, B_2$ and $s : B_{i\varepsilon} \vdash_p A$. Hence, we have $\pi_i^\varepsilon \cdot s : B_1 \,\&\, B_2 \vdash_p A$ and so $\langle v \mid \pi_i^\varepsilon \cdot s \mid l \rangle^- : A$.
- $\langle v \mid (x^+.t)^\varepsilon \cdot s \mid l \rangle^+ \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle^\varepsilon$: 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 $\langle t[v/x] \mid s \mid l \rangle^\varepsilon : A$.
- $\langle \lambda x.t \mid v^\varepsilon \cdot s \mid l \rangle^- \rightsquigarrow \langle t[v/x] \mid s \mid l \rangle^\varepsilon$: by hypothesis, we get $x : B \vdash_p t : C_\varepsilon, \vdash_p v : B$ and $s : C \vdash_p A$. By SL, we have $\vdash_p t[v/x] : C$ and so $\langle t[v/x] \mid s \mid l \rangle^\varepsilon : A$.
- $\langle \langle t_1, t_2 \rangle \mid \pi_i^\varepsilon \cdot s \mid l \rangle^- \rightsquigarrow \langle t_i \mid s \mid l \rangle^\varepsilon$: by hypothesis, we get $\vdash_p t_1 : B_1, \vdash_p t_2 : B_2$ and $s : B_{i\varepsilon} \vdash_p A$. Hence, we have $\langle t_i \mid s \mid l \rangle^\varepsilon : A$.
- $\langle \delta((v, w), (x, y).t)^\varepsilon \mid s \mid l \rangle^\varepsilon \rightsquigarrow \langle t[v/x, w/y] \mid s \mid l \rangle^\varepsilon$: by hypothesis, we get $\vdash_p v : B, \vdash_p w : C, x : B, y : C \vdash_p t : D_\varepsilon$ 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 $\langle t[v/x, w/y] \mid s \mid l \rangle^\varepsilon : A$.
- $\langle \delta((), ().t)^\varepsilon \mid s \mid l \rangle^\varepsilon \rightsquigarrow \langle t \mid s \mid l \rangle^\varepsilon$: by hypothesis, we get $\vdash_p t : B_\varepsilon$ and $s : B \vdash_p A$, hence we have $\langle t \mid s \mid l \rangle^\varepsilon : A$.
- $\langle \delta(\iota_i v, x_1.t_1, x_2.t_2)^\varepsilon \mid s \mid l \rangle^\varepsilon \rightsquigarrow \langle t_i[v/x_i] \mid s \mid l \rangle^\varepsilon$: by hypothesis, we get $\vdash_p v : B_i, x_1 : B_1 \vdash_p t_1 : C_\varepsilon, x_2 : B_2 \vdash_p t_2 : C$ and $s : C \vdash_p A$. By SL, we have $\vdash_p t_i[v/x_i] : C$ and so $\langle t_i[v/x_i] \mid s \mid l \rangle^\varepsilon : A$.
- $\langle \textbf{new} \mid () \cdot s \mid r_n :: l \rangle^- \rightsquigarrow \langle \iota_1 r_n \mid s \mid l \rangle^+$: by hypothesis, we have $s : R \oplus 1 \vdash_p A$. We always have $\vdash_p \iota_1 r_n : R \oplus 1$, and so $\langle \iota_1 r_n \mid s \mid l \rangle^+ : A$.
- $\langle \textbf{new} \mid () \cdot s \mid \text{Nil} \rangle^- \rightsquigarrow \langle \iota_2() \mid s \mid \text{Nil} \rangle^+$: by hypothesis, we have $s : R \oplus 1 \vdash_p A$. We always have $\vdash_p \iota_2() : R \oplus 1$, and so $\langle \iota_1 r_n \mid s \mid l \rangle^+ : A$.
- $\langle \textbf{delete} \mid r_n \cdot s \mid l \rangle^- \rightsquigarrow \langle () \mid s \mid r_n :: l \rangle^+$: 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 5)

Show that any well-typed command $\langle t \mid s \mid l \rangle^\varepsilon : A$ reduces iff it is not of the shape $\langle v_t \mid \star \mid l \rangle^\varepsilon$ by case analysis on the judgement $\vdash_p t : B_\varepsilon$:
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 $\langle v_t \mid \star \mid l \rangle^\varepsilon$ and matches no reduction steps. Otherwise, the stack shape is determined by the type of t:

- If $\varpi(B) = +$, then $s = (x^+.u)^\varepsilon \cdot s'$ and $\langle v_t \mid (x^+.u)^\varepsilon \cdot s' \mid l \rangle^+ \rightsquigarrow \langle u[v_t/x] \mid s' \mid l \rangle^\varepsilon$.

- If $t = \langle t_1, t_2 \rangle$, then $s = \pi_i^\varepsilon \cdot s'$ and $\langle \langle t_1, t_2 \rangle \mid \pi_i^\varepsilon \cdot s' \mid l \rangle^- \rightsquigarrow \langle t_i \mid s' \mid l \rangle^\varepsilon$.
- If $t = \lambda x.t'$, then $s = v^\varepsilon \cdot s'$ and $\langle \lambda x.t' \mid v^\varepsilon \cdot s' \mid l \rangle^- \rightsquigarrow \langle t[v/x] \mid s' \mid l \rangle^\varepsilon$.
- If $t = \mathbf{new}$ and $l = r_n :: l'$, then $s = () \cdot s'$ and $\langle \mathbf{new} \mid () \cdot s' \mid r_n :: l' \rangle^- \rightsquigarrow \langle \iota_1 r_n \mid s' \mid l' \rangle^+$.
- If $t = \mathbf{new}$ and $l = \star$, then $s = () \cdot s'$ and $\langle \mathbf{new} \mid () \cdot s' \mid \star \rangle^- \rightsquigarrow \langle \iota_2() \mid s' \mid \star \rangle^+$.
- If $t = \mathbf{delete}$ so $s = r_n \cdot s'$ and $\langle \mathbf{delete} \mid r_n \cdot s' \mid l \rangle^- \rightsquigarrow \langle () \mid s' \mid r_n :: l \rangle^+$.

Otherwise:

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

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

C Details: resource-safety properties

C.1 Proof that $\vdash$ implies resource-free $\vdash_o$ (lemma 7)

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 8)

Show that (Left SL) if $; L_t; \vdash_o t$ and $\Gamma, x; L_u; \Delta \vdash_o u$, 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:

- $(), \mathbf{new}, \mathbf{delete}, r_n$: 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 $\Theta \mathbin{@} \Theta'$ is well-formed. In the left SL case:
  - $\Theta = \Gamma_v;;$ and $\Theta' = \Gamma^1_w, x; L_u; \Gamma^2_w$: x does not occur in v so $(v, w)[t/x] = (v, w[t/x])$ and by IH on w we have $\Gamma^1_w, x; L_u; \Gamma^2_w \vdash_o w[t/x]$, hence we have $\Gamma_v, \Gamma^1_w; L_t +\!\!+ L_u; \Gamma^2_w \vdash_o (v, w)[t/x]$.
  - $\Theta = \Gamma^1_v, x; L_u; \Gamma^2_v$ and $\Theta' = \Gamma_w;;:$ $(v, w)[t/x] = (v[t/x], w)$ so the SL holds by IH on $\Theta \vdash_o v$.
  - $\Theta = \Gamma_v, x; L^1_u;$ and $\Theta' =; L^2_u; \Gamma_w$: $(v, w)[t/x] = (v[t/x], w)$ so the SL holds by IH on v.

  The right SL is done similarly, swapping x and L_u:
  - $\Theta =;; \Gamma_v$ and $\Theta' = \Gamma^1_w; L_u; x, \Gamma^2_w$: $(v, w)[t/x] = (v, w[t/x])$ so the SL holds by IH on w.
  - $\Theta = \Gamma^1_v; L_u; x, \Gamma^2_v$ and $\Theta' = \Gamma_w$: $(v, w)[t/x] = (v[t/x], w)$ so the SL holds by IH on v.
  - $\Theta = \Gamma_v; L^1_u;$ and $\Theta' =; L^2_u; x, \Gamma_w$: $(v, w)[t/x] = (v, w[t/x])$ 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 $\Theta \mathbin{@} \Theta'$. For the left SL:
  - $\Theta =;; \Gamma_{u2}, y$ and $\Theta' = \Gamma^1_{u1}, x, ; L_u; \Gamma^2_{u1}$: $(\text{let } y = u_1 \text{ in } u_2)[t/x] = (\text{let } y = u_1 \text{ in } u_2[t/x])$ so the SL holds by IH on u_2.
  - $\Theta = \Gamma^1_{u2}, x; L_u; \Gamma^2_{u2}, y$ and $\Theta' =;; \Gamma_w$: $(\text{let } y = u_1 \text{ in } u_2)[t/x] = (\text{let } y = u_1[t/x] \text{ in } u_2)$ so the SL holds by IH on u_1.
  - $\Theta = \Gamma_{u2}; L^1_u; y$ and $\Theta' =; L^2_u; x, \Gamma_{u1}$: 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 $\Theta \mathbin{@} \Theta' \mathbin{@} \Theta''$, here for left SL:
  - $\Theta = \Gamma^1_u, x; L_u; \Gamma^2_u, \Theta' =;; \Gamma_v, \Theta'' =;; \Gamma^3_u$: $(v, y.u_1, z.u_2)[t/x] = (v, y.u_1[t/x], z.u_2[t/x])$ so the SL holds by IH on $\Gamma^1_u, x; L_u; \Gamma^2_u, y, \Gamma^3_u \vdash_o u_1$ and $\Gamma^1_u, x; L_u; \Gamma^2_u, y, \Gamma^3_u \vdash_o u_2$.
  - $\Theta = \Gamma^1_u, x; L^1_u, \Theta' =; L^2_u; \Gamma_v, \Theta'' =;; \Gamma^2_u$: the SL holds by IH on u_1 and u_2.
  - $\Theta =;; \Gamma^1_u, \Theta' = \Gamma^1_v, x; L_u; \Gamma^2_v, \Theta'' =;; \Gamma^2_u$: the SL holds by IH on v.
  - $\Theta =;; \Gamma^1_u, \Theta' = \Gamma^1_v, x; L^1_u; \Theta'' =; L^2_u; \Gamma^2_u$: the SL holds by IH on v.
  - $\Theta =;; \Gamma^1_u, \Theta' =;; \Gamma_v, \Theta'' = \Gamma^2_u, x; L_u; \Gamma^3_u$: 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 $\Theta \mathbin{@} \Theta' \mathbin{@} \Theta''$.
- $\delta(v, ().t)$: by case analysis on $\Theta \mathbin{@} \Theta'$.
- vw: by case analysis on $\Theta \mathbin{@} \Theta'$.

C.3 Proof of ordered predicate invariant by reduction (theorem 9)

Show that $\forall c1, \forall c2, \forall L, L \vdash^C_o c1 \wedge c1 \rightsquigarrow c2 \Rightarrow L \vdash^C_o c2$ 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 $L \vdash^C_o c1$, we get $; L_t; \vdash_o t, ; L_u; x \vdash_o u$ and $L_s \vdash^S_o s$ with $L = L_s +\!\!+ L_u +\!\!+ L_t +\!\!+ l$, hence we have $; L_s +\!\!+ L_u; \vdash_o (x^+.u)\cdot s$ and so $L_s +\!\!+ L_u +\!\!+ L_t +\!\!+ l = L \vdash^C_o c2$.
- $\langle \text{let } x^- = t \text{ in } u \mid s \mid l\rangle \rightsquigarrow \langle u[t/x] \mid s \mid l\rangle$: by definition, we get $; L_t; \vdash_o t, ; L_u; x \vdash_o u$ and $L_s \vdash^S_o s$ hence by right SL we have $; L_u +\!\!+ L_t; \vdash_o u[t/x]$ and so $L \vdash^C_o c2$.
- $\langle vw \mid s \mid l\rangle \rightsquigarrow \langle v \mid w\cdot s \mid l\rangle$: by definition, we get $; L_w; \vdash_o w, ; L_v; \vdash_o v$ and $L_s \vdash^S_o s$, hence we have $L_s +\!\!+ L_w \vdash^S_o w\cdot s$ and so $L \vdash^C_o c2$.
- $\langle \pi_i t \mid s \mid l\rangle \rightsquigarrow \langle t \mid \pi_i \cdot s \mid l\rangle$: by definition, we get $; L_t; \vdash_o t$ and $L_s \vdash^S_o s$, hence we have $L \vdash^C_o c2$.
- $\langle v \mid (x.t)\cdot s \mid l\rangle \rightsquigarrow \langle t[v/x] \mid s \mid l\rangle$: by definition, we get $; L_v; \vdash_o v, ; L_t; x \vdash_o t$ and $L_s \vdash^S_o s$, hence by right SL we have $; L_t +\!\!+ L_v; \vdash_o t[v/x]$ and so $L \vdash^C_o c2$.
- $\langle \lambda x.t \mid v\cdot s \mid l\rangle \rightsquigarrow \langle t[v/x] \mid s \mid l\rangle$: by definition, we get $L_v \vdash_o v, x, L_t \vdash_o t$ and $L_s \vdash^S_o s$, hence by left SL we have $L_v, L_t \vdash_o t[v/x]$ and so $L \vdash^C_o c2$.
- $\langle \langle t_1, t_2\rangle \mid \pi_i \cdot s \mid l\rangle \rightsquigarrow \langle t_i \mid s \mid l\rangle$: by definition, we get $L_t \vdash_o t_1$, $L_t \vdash_o t_2$ and $L_s \vdash^S_o s$, hence we have $L \vdash^C_o c2$.
- $\langle \delta((v,w),(x,y).t) \mid s \mid l\rangle \rightsquigarrow \langle t[v/x, w/y] \mid s \mid l\rangle$: 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 $L_s \vdash^S_o s$ with $\Theta =$ either $; L_t; x, y$ or $x, y; L_t;$. In the first case, by right SL twice we have $; L_t +\!\!+ L_v +\!\!+ L_w; \vdash_o t[v/x, w/y]$ and so $L \vdash^C_o c2$. Similarly, in the second case by left SL twice we have $; L_v +\!\!+ L_w +\!\!+ L_t; \vdash_o t[v/x, w/y]$ and so $L \vdash^C_o c2$.
- $\langle \delta((),().t) \mid s \mid l\rangle \rightsquigarrow \langle t \mid s \mid l\rangle$: by definition, we get $; []; \vdash_o (), ; L_t; \vdash_o t$ and $L_s \vdash^S_o s$, hence we have $L \vdash^C_o c2$.
- $\langle \delta(\iota_i v, x_1.t_1, x_2.t_2) \mid s \mid l\rangle \rightsquigarrow \langle t_i[x_i/v] \mid s \mid l\rangle$: by definition and case analysis, we get $; L_v; \vdash_o v, \Theta_i \vdash t_i$ and $L_s \vdash^S_o s$ with both $\Theta_i =$ either $; L_t; x_i$ or $x_i; L_t;$. In the first case, by right SL we have $; L_t +\!\!+ L_v; \vdash_o t_i[v/x_i]$ and so $L \vdash^C_o c2$. Similarly, in the second case by left SL we have $; L_v +\!\!+ L_t; \vdash_o t_i[v/x_i]$ and so $L \vdash^C_o c2$.
- $\langle \textbf{new} \mid () \cdot s \mid r_n :: l\rangle \rightsquigarrow \langle \iota_1 r_n \mid s \mid l\rangle$: by definition, we get $; []; \vdash_o (), ; [r_n]; \vdash_o r_n$ and $L_s \vdash^S_o s$, hence we have $L_s +\!\!+ r_n :: l = L \vdash^C_o c2$.
- $\langle \textbf{new} \mid () \cdot s \mid \text{Nil}\rangle \rightsquigarrow \langle \iota_2() \mid s \mid \text{Nil}\rangle$: by definition, we get $L_s = L \vdash^C_o c2$.
- $\langle \textbf{delete} \mid r_n \cdot s \mid l\rangle \rightsquigarrow \langle () \mid s \mid r_n :: l\rangle$: by definition, we get $; [r_n]; \vdash_o r_n, ; []; \vdash_o ()$ and $L_s \vdash^S_o s$, hence we have $L_s +\!\!+ r_n :: l = L \vdash^C_o c2$.

C.4 Proof of (theorem 10) 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: $\Theta = (M_1; \Gamma_1) \mathbin{@} \Theta' = (M_2; \Gamma_2) \stackrel{\text{def}}{=} M_1, M_2; \Gamma_1, \Gamma_2$.

We can then prove the following substitution lemma:

Lemma 18. *If $\Theta \vdash_l t$ and $\Theta', x \vdash_l u$, then $\Theta \mathbin{@} \Theta' \vdash_l u[t/x]$.*

Proof. By induction on u:

- $()$, **new**, **delete**, r_n: by absurdity from the induction hypothesis (IH): no variables in the context.

$$\frac{}{[];x\vdash_l x}\qquad\frac{}{[];\vdash_l ()}\qquad\frac{}{[];\vdash_l \mathbf{new}}\qquad\frac{}{[];\vdash_l \mathbf{delete}}\qquad\frac{}{[r_n];\vdash_l r_n}$$

$$\frac{\Theta\vdash_l v}{\Theta\vdash_l \iota_i v}\qquad\frac{\Theta\vdash_l v}{\Theta\vdash_l \pi_i v}\qquad\frac{\Theta\vdash_l t\qquad\Theta\vdash_l u}{\Theta\vdash_l \langle t,u\rangle}\qquad\frac{\Theta,x\vdash_l t}{\Theta\vdash_l \lambda x.t}\qquad\frac{\Theta\vdash_l v\qquad\Theta'\vdash_l w}{\Theta\mathbin{@}\Theta'\vdash_l (v,w)}$$

$$\frac{\Theta,x\vdash_l u\qquad\Theta'\vdash_l t}{\Theta\mathbin{@}\Theta'\vdash_l \text{let } x=t \text{ in } u}\qquad\frac{\Theta,x,y\vdash_l t\qquad\Theta'\vdash_l v}{\Theta\mathbin{@}\Theta'\vdash_l \delta(v,(x,y).t)}\qquad\frac{\Theta\vdash_l t\qquad\Theta'\vdash_l v}{\Theta\mathbin{@}\Theta'\vdash_l \delta(v,().t)}$$

$$\frac{\Theta,x\vdash_l t\qquad\Theta,y\vdash_l u\qquad\Theta'\vdash_l v}{\Theta\mathbin{@}\Theta'\vdash_l \delta(v,x.t,y.u)}\qquad\frac{\Theta\vdash_l w\qquad\Theta'\vdash_l v}{\Theta\mathbin{@}\Theta'\vdash_l vw}$$

Fig. 7. Predicate of linear expressions with resources

$$\frac{}{\vdash_l^S \star}\qquad\frac{M_v;\vdash_l v\qquad M_s\vdash_l^S s}{M_s+\!\!+M_v\vdash_l^S v\cdot s}\qquad\frac{M_t;x\vdash_l t\qquad M_s\vdash_l^S s}{M_s+\!\!+M_s\vdash_l^S (x^+.t)\cdot s}$$

$$\frac{M_t;\vdash_l t\qquad M_s\vdash_l^S s}{M_t+\!\!+M_s+\!\!+l\vdash_l^C \langle t\mid s\mid l\rangle}$$

Fig. 8. Typing rules of ordered stacks and commands

- $;x;\vdash_l x$: 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 $\Theta\mathbin{@}\Theta'$, 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:
  - $x\in\Theta$: $(\text{let } y=u_1 \text{ in } u_2)[t/x]=(\text{let } y=u_1 \text{ in } u_2[t/x])$ so the SL holds by IH on $\Theta,y\vdash_l u_2$.
  - $x\in\Theta'$: $(\text{let } y=u_1 \text{ in } u_2)[t/x]=(\text{let } y=u_1[t/x] \text{ in } u_2)$ so the SL holds by IH on $\Theta'\vdash_l u_1$.
- $(v,y.u_1,z.u_2)$: 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.
- 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 19. *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 $c1 \rightsquigarrow c2$:

- $\langle \text{let } x^+ = t \text{ in } u \mid s \mid l\rangle \rightsquigarrow \langle t \mid (x^+.u) \cdot s \mid l\rangle$: by definition of $M \vdash_l^C c1$, we get $M_t; \vdash_l t$, $M_u; x \vdash_l u$ and $M_s \vdash_l^S s$ with $M = M_s +\!\!+ M_u +\!\!+ M_t +\!\!+ l$, hence we have $M_s +\!\!+ M_u \vdash_l (x^+.u) \cdot s$ and so $M_s +\!\!+ M_u +\!\!+ M_t +\!\!+ l = M \vdash_l^C c2$.
- $\langle \text{let } x^- = t \text{ in } u \mid s \mid l\rangle \rightsquigarrow \langle u[t/x] \mid s \mid l\rangle$: by definition, we get $M_t; \vdash_l t$, $M_u; x \vdash_l u$ and $M_s \vdash_l^S s$, hence by SL we have $M_t +\!\!+ M_u; \vdash_l t[u/x]$ and so $M \vdash_l^C c2$.
- $\langle \pi_i t \mid s \mid l\rangle \rightsquigarrow \langle t \mid \pi_i \cdot s \mid l\rangle$: by definition, we get $M_t; \vdash_l t$ and $M_s \vdash_l^S s$, hence $M \vdash_l^C c2$.
- $\langle v \mid (x.t) \cdot s \mid l\rangle \rightsquigarrow \langle t[v/x] \mid s \mid l\rangle$: by definition, we get $M_v; \vdash_l v$, $M_t; x \vdash_l t$ and $M_s \vdash_l^S s$, hence by SL we have $M_t +\!\!+ M_v; \vdash_l t[v/x]$ and so $M \vdash_l^C c2$.
- $\langle \lambda x.t \mid v \cdot s \mid l\rangle \rightsquigarrow \langle t[v/x] \mid s \mid l\rangle$: by definition, we get $M_t; x \vdash_l t$, $M_v; \vdash_l v$ and $M_s \vdash_l^S s$, hence by SL we have $M_t +\!\!+ M_v; \vdash_l t[v/x]$ and so $M \vdash_l^C c2$.
- $\langle \langle t_1, t_2\rangle \mid \pi_i \cdot s \mid l\rangle \rightsquigarrow \langle t_i \mid s \mid l\rangle$: by definition, we get $M_t; \vdash_l t_1$, $M_t; \vdash_l t_2$ and $M_s \vdash_l^S s$, hence $M \vdash_l^C c2$.
- $\langle \delta((v, w), (x, y).t) \mid s \mid l\rangle \rightsquigarrow \langle t[v/x, w/y] \mid s \mid l\rangle$: by definition, 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 $M_t +\!\!+ M_v +\!\!+ M_w; \vdash_l t[v/x][w/y] = t[v/x, w/y]$ and so $M \vdash_l^C c2$.
- $\langle \delta((), ().t) \mid s \mid l\rangle \rightsquigarrow \langle t \mid s \mid l\rangle$: by definition, we get $M_t; \vdash_l t$ and $M_s \vdash_l^S s$, hence $M \vdash_l^C c2$.
- $\langle \delta(\iota_i v, x_1.t_1, x_2.t_2) \mid s \mid l\rangle \rightsquigarrow \langle t_i[x_i/v] \mid s \mid l\rangle$: 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 $M_t +\!\!+ M_v; \vdash_l t_i[v/x_i]$ and so $M \vdash_l^C c2$.
- $\langle \textbf{new} \mid () \cdot s \mid r_n :: l\rangle \rightsquigarrow \langle \iota_1 r_n \mid s \mid l\rangle$: by definition, we get $; \vdash_l \textbf{new}$, $; \vdash_l ()$ and $M_s \vdash_l^S s$, hence $M_s +\!\!+ r_n :: l = M \vdash_l^C c2$.
- $\langle \textbf{new} \mid () \cdot s \mid \text{Nil}\rangle \rightsquigarrow \langle \iota_2() \mid s \mid \text{Nil}\rangle$: by definition, we get $; \vdash_l \textbf{new}$, $; \vdash_l ()$ and $M_s \vdash_l^S s$, hence $M_s = M \vdash_l^C c2$.
- $\langle \textbf{delete} \mid r_n \cdot s \mid l\rangle \rightsquigarrow \langle () \mid s \mid r_n :: l\rangle$: by definition, we get $; \vdash_l \textbf{delete}$, $r_n; \vdash_l r_n$ and $M_s \vdash_l^S s$, hence $M_s +\!\!+ r_n :: l = M \vdash_l^C c2$.

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 Details: translation of the resource CBPV

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 $[\![\vdash \text{Alloc\_failure} : E]\!] = \vdash \text{Alloc\_failure} : E$. Moreover, because $\mathcal{O}_{\mathcal{E},\mathbf{move}}$ is an extension of $\mathcal{L}$, it can use linear swaps for exceptions.

We proceed by cases:

- $[\![x \vdash x : A]\!]$ must be a value of type $A^+ \vdash A^+$. Hence, x is well-typed.
- $[\![\vdash \mathbf{drop} : 1 \multimap A]\!]$ must be a value of type $\vdash (1 \multimap A^+) \,\&\, 1$. Hence, $\langle \text{drop}_A, () \rangle$ is well-typed.
- $[\![\Gamma \vdash \text{coerc}(v) : P]\!]$ must be a value of type $\Gamma^+ \vdash (P^+ \oplus E) \,\&\, I$. By IH, we have $\Gamma^+ \vdash [\![v]\!] : P^+$. Hence, $\langle \iota_1 [\![v]\!], \text{drop\_ctx}_\Gamma \rangle$ is well-typed.
- $[\![\Gamma, x : A, \Delta \vdash \mathbf{move}(x) \text{ in } t : C]\!]$ must be a value of type $\Gamma^+, \Delta^+, x : A^+ \vdash \Downarrow C^-$. By IH, we have $\Gamma^+, x : A^+, y : B^+, \Delta^+ \vdash [\![t]\!] : \Downarrow C^-$. It's the only rule where we allow ourselves to use $\mathcal{L}$ structural rule to swap premises. Hence, $[\![t]\!]$ is well-typed.
- $[\![\vdash \mathbf{new} : R \multimap 1]\!]$ must be a value of type $\vdash R \oplus E \multimap 1$. We have $\vdash \text{Alloc\_failure} : E$, so let $x = \mathbf{new}()$ in $\delta(x, r.\iota_1 r, i.i; \iota_2 \text{Alloc\_failure})$ is well-typed.
- $[\![\Gamma, \Delta \vdash \text{let } x = (v : A) \text{ in } t : B]\!]$ must be a value of type $\Gamma^+, \Delta^+ \vdash \Downarrow B^-$. By IH, we have $\Delta^+ \vdash [\![v]\!] : A^+$ and $\Gamma^+, x : A^+ \vdash [\![t]\!] : \Downarrow B^-$. Hence, let $x = [\![v]\!]$ in $[\![t]\!]$ is well-typed.
- $[\![\Gamma, \Delta \vdash \text{let } x = (t : P) \text{ in } u : A]\!]$ must be a value of type $\Gamma^+, \Delta^+ \vdash A^- \,\&\, I$. By IH, we have $\Delta^+ \vdash [\![t]\!] : (P^+ \oplus E) \,\&\, I$ and $\Gamma^+, x : P^+ \vdash [\![u]\!] : A^- \,\&\, I$. Then, $\text{raise}_\Gamma^{A \& I}(e)$ is of type $\Gamma^+ \vdash A^- \,\&\, I$, so let $s = \pi_1 [\![t]\!]$ in $\delta(s, x.[\![u]\!], e.\, \text{raise}_\Gamma^{A \& I}(e))$ is well-typed.
- $[\![\vdash \mathbf{raise} : A \multimap E]\!]$ must be a value of type $\vdash (A^- \multimap E) \& I$. Hence, $\langle \lambda e.\, \text{raise}_\star^A(e), () \rangle$ is well-typed.
- $[\![\Gamma, \Delta \vdash \mathbf{try}\, x \Leftarrow (t : P) \text{ in } u \,\mathbf{unless}\, e \Rightarrow u' : B]\!]$ must be a value of type $\Gamma^+, \Delta^+ \vdash \Downarrow B^-$. By IH, we have:
  - $\Delta^+ \vdash [\![t]\!] : (P^+ \oplus E) \,\&\, I$.
  - $\Gamma^+, x : P^+ \vdash [\![u]\!] : \Downarrow B^-$.
  - $\Gamma^+, e : E \vdash [\![u']\!] : \Downarrow B^-$.

  Hence, let $s = \pi_1 [\![t]\!]$ in $\delta(s, x.[\![u]\!], e.[\![u']\!])$ is well-typed.
- $[\![\Gamma, \Delta \vdash (v, w) : A \otimes B]\!]$ must be a value of type $\Gamma^+, \Delta^+ \vdash A^+ \otimes B^+$. By IH, we have $\Gamma^+ \vdash [\![v]\!] : A^+$ and $\Delta^+ \vdash [\![w]\!] : B^+$. Hence, $([\![v]\!], [\![w]\!])$ is well-typed.
- $[\![\Gamma, \Delta, \Gamma' \vdash \delta(v, (x, y).t) : C]\!]$ must be a value of type $\Gamma^+, \Delta^+, \Gamma'^+ \vdash \Downarrow C^-$. By IH, we have $\Delta^+ \vdash [\![v]\!] : A^+ \otimes B^+$ and $\Gamma^+, x : A^+, y : B^+, \Gamma'^+ \vdash [\![t]\!] : \Downarrow C^-$. Hence, $\delta([\![v]\!], (x, y).[\![t]\!])$ is well-typed.
- $[\![\vdash () : 1]\!]$ must be a value of type $\vdash 1$. Hence, $()$ is well-typed.
- $[\![\Gamma, \Delta, \Gamma' \vdash \delta(v, ().t) : C]\!]$ must be a value of type $\Gamma^+, \Delta^+, \Gamma'^+ \vdash \Downarrow C^-$. By IH, we have $\Delta^+ \vdash [\![v]\!] : 1$ and $\Gamma^+, \Gamma'^+ \vdash [\![t]\!] : \Downarrow C^-$. Hence, $\delta([\![v]\!], ().[\![t]\!])$ is well-typed.

- $[\![\Gamma \vdash \iota_1 v : A \oplus B]\!]$ must be a value of type $\Gamma^+ \vdash A^+ \oplus B^+$. By IH, we have $\Gamma^+ \vdash [\![v]\!] : A^+$. Hence, $\iota_1 [\![v]\!]$ is well-typed.
- $[\![\Gamma \vdash \iota_2 v : A \oplus B]\!]$ must be a value of type $\Gamma^+ \vdash A^+ \oplus B^+$. By IH, we have $\Gamma^+ \vdash [\![v]\!] : B^+$. Hence, $\iota_2 [\![v]\!]$ is well-typed.
- $[\![\Gamma, \Delta, \Gamma' \vdash \delta(v, x.t, y.u) : C]\!]$ must be a value of type $\Gamma^+, \Delta^+, \Gamma'^+ \vdash \Downarrow C^-$. By IH, we have:
  - $\Delta^+ \vdash [\![v]\!] : A^+ \oplus B^+$.
  - $\Gamma^+, x : A^+, \Gamma'^+ \vdash [\![t]\!] : \Downarrow C^-$.
  - $\Gamma^+, y : B^+, \Gamma'^+ \vdash [\![u]\!] : \Downarrow C^-$.

  Hence, $\delta([\![v]\!], x.[\![t]\!], y.[\![u]\!])$ is well-typed.
- $[\![\Gamma \vdash \lambda x.t : B \multimap A]\!]$ must be a value of type $\Gamma^+ \vdash (B^- \multimap A^+) \,\&\, I$. By IH, we have $x : A^+, \Gamma^+ \vdash [\![t]\!] : B^- \,\&\, I$. Hence, $\langle \lambda x.\pi_1 [\![t]\!], \mathrm{drop\_ctx}_\Gamma \rangle$ is well-typed.
- $[\![\Gamma, \Delta \vdash vw : B]\!]$ must be a value of type $\Gamma^+, \Delta^+ \vdash \Downarrow B^-$. By IH, we have $\Gamma^+ \vdash [\![w]\!] : A^+$ and $\Gamma^+ \vdash [\![v]\!] : (B^- \multimap A^+) \,\&\, I$. Hence, $\langle (\pi_1 [\![v]\!])[\![w]\!], \mathrm{drop\_ctx}_{\Gamma,\Delta} \rangle$ is well-typed.
- $[\![\Gamma \vdash \langle t, u \rangle : A \,\&\, B]\!]$ must be a value of type $\Gamma^+ \vdash (A^- \,\&\, B^-) \,\&\, I$. By IH, we have $\Gamma^+ \vdash [\![t]\!] : A^- \,\&\, I$ and $\Gamma^+ \vdash [\![u]\!] : B^- \,\&\, I$. Hence, $\langle \langle \pi_1 [\![t]\!], \pi_1 [\![u]\!] \rangle, \mathrm{drop\_ctx}_\Gamma \rangle$ is well-typed.
- $[\![\Gamma \vdash \pi_1 v : A]\!]$ must be a value of type $\Gamma^+ \vdash A^- \,\&\, I$. By IH, we have $\Gamma^+ \vdash [\![v]\!] : (A^- \,\&\, B^-) \,\&\, I$. Hence, $\langle \pi_1 \pi_1 [\![v]\!], \mathrm{drop\_ctx}_\Gamma]\!] \rangle$ is well-typed.
- $[\![\Gamma \vdash \pi_2 v : B]\!]$ must be a value of type $\Gamma^+ \vdash B^- \,\&\, I$. By IH, we have $\Gamma^+ \vdash [\![v]\!] : (A^- \,\&\, B^-) \,\&\, I$. Hence, $\langle \pi_2 \pi_1 [\![v]\!], \mathrm{drop\_ctx}_\Gamma]\!] \rangle$ is well-typed.

E Details: presheaf model

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

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 $\Gamma \otimes FA \cong F(\Gamma \otimes A)$ in the relevant direction. $f^{\lambda^{\circ-/-\circ}}$ 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 $\Gamma \otimes (A \oplus B) \otimes \Gamma' \to (\Gamma \otimes A \otimes \Gamma') \oplus (\Gamma \otimes B \otimes \Gamma')$ 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 $\mathrm{coerc}(v)$ below.

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

Stacks are interpreted as such:

- $[\![[]; \star : A \vdash_o A]\!] \stackrel{\text{def}}{=} id_{A^-} : \mathscr{C}(A^-, A^-)$
- $[\![Ls; \pi_1 \cdot s : A \,\&\, B \vdash_o C]\!] \stackrel{\text{def}}{=} \pi_1; [\![s]\!] : \mathscr{C}(Ls^+ \otimes (A^- \,\&\, B^-), C^-)$
- $[\![Ls; \pi_2 \cdot s : A \,\&\, B \vdash_o C]\!] \stackrel{\text{def}}{=} \pi_2; [\![s]\!] : \mathscr{C}(Ls^+ \otimes (A^- \,\&\, B^-), C^-)$
- $[\![Ls +\!\!+ Lt; v \cdot s : B \circ- A \vdash_o C]\!] \stackrel{\text{def}}{=} (id_{Ls^+} \otimes (([\![v]\!] \otimes id_{B^- \circ- A^+}); ev^{\circ-})); [\![s]\!] : \mathscr{C}(Ls^+ \otimes Lt^+ \otimes (B^- \circ- A^+), C^-)$
- $[\![Ls +\!\!+ Lt; (x^+.t) \cdot s : A \vdash_o C]\!] \stackrel{\text{def}}{=} (id_{Ls^+} \otimes [\![t]\!]^{*;str_{Lt^+}}); [\![s]\!] : \mathscr{C}(Ls^+ \otimes Lt^+ \otimes FA^+, C^-)$

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

E.2 Proof of (lemma 13)

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 $f;(g^*;h)^* = (Ff;(g^*;h))^* = ((f;g)^*;h)^*$:

- $[\![\mathrm{coerc}(w)[v/x]]\!] = [\![w[v/x]]\!];\eta = [\![\mathrm{coerc}(w[v/x])$ by induction hypothesis.
- $[\![x[v/x]]\!] = [\![v]\!]$.
- No variables in r, **new** nor ().
- $[\![\Gamma, \Delta \vdash_o (v, w)]\!]$: either $x \in \Gamma$ or $x \in \Delta$, in which cases we need to show $[\![(w, w')[v/x]]\!] = [\![(w[v/x], w')]\!]$ or $[\![(w[v/x], w')]\!]$. Those hold by IH.
- $[\![(\iota_i w)[v/x]]\!] = [\![\iota_i(w[v/x])]\!]$ by IH.
- $[\![\delta(w, ().t)[v/x]]\!]$: split case on $v \in \Gamma, v \in \Delta$ or $v \in \Gamma'$. In each case, the SL holds by IH.
- $[\![\delta(w, (y, z).t)[v/x]]\!]$: same split case. In each case, the SL holds by IH.
- $[\![\delta(w, y.t, z.u)[v/x]]\!]$: 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 $(id_{\Gamma+} \otimes [\![v]\!] \otimes id_{\Gamma'+}); ([\![t]\!]^* \times [\![u]\!]^*)^* = (((id_{\Gamma+} \otimes [\![v]\!] \otimes id_{\Gamma'+}); [\![t]\!])^* \times ((id_{\Gamma+} \otimes [\![v]\!] \otimes id_{\Gamma'+}); [\![u]\!])^*)^*$. 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.
- $[\![\Gamma, \Delta \vdash_o (\mathrm{let}\ y = (t : P)\ \mathrm{in}\ u)[v/x]]\!]$: split case on $v \in \Gamma$ or $v \in \Delta$, in both cases it holds by SL and naturality of $*$ and str.
- $[\![\Gamma, \Delta \vdash_o (\mathrm{let}\ y = (t : N)\ \mathrm{in}\ u)[v/x]]\!]$: same split case. In both cases it holds by SL.

E.3 Proof of (theorem 14)

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 $(Ls^+ \otimes [\![t]\!]^*); [\![s]\!]$ from both command are equal. Finally, we consider the whole command interpretation pontwise for **new** and **delete** reduction rules.

- $\langle \mathrm{let}\ x^- = v\ \mathrm{in}\ u \mid s \mid l\rangle \rightsquigarrow \langle u[t/x] \mid s \mid l\rangle$: we need to show that $[\![\mathrm{let}\ x^- = v\ \mathrm{in}\ u]\!] = [\![u[v/x]]\!]$. Both expressions are equal by SL and definition of binding a negative value, that is precomposition.
- $\langle \mathrm{let}\ x^+ = t\ \mathrm{in}\ u \mid s \mid l\rangle \rightsquigarrow \langle t \mid (x^+.u) \cdot s \mid l\rangle$: we need to show that $(id_{Ls^+} \otimes ((id_{Lu^+} \otimes [\![t]\!]^*); [\![u]\!]^{*;str_{Lu^+}})^{str_{Lu^+};*;*}); [\![s]\!] = (id_{Ls^+} \otimes id_{Lu^+} \otimes [\![t]\!]^*); (id_{Ls^+} \otimes [\![u]\!]^{*;str_{Lu^+}}); [\![s]\!]$. 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.
- $\langle v \mid (x.t) \cdot s \mid l\rangle \rightsquigarrow \langle t[v/x] \mid s \mid l\rangle$: $[\![\mathrm{coerc}(v)]\!]^* = F[\![v]\!]$, 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).
- $\langle \mathbf{new} \mid () \cdot s \mid r_n :: l \rangle \rightsquigarrow \langle \iota_1 r_n \mid s \mid l \rangle$: by definition of popping elements from a non-empty list of resources.
- $\langle \mathbf{new} \mid () \cdot s \mid \mathrm{Nil} \rangle \rightsquigarrow \langle \iota_2() \mid s \mid \mathrm{Nil} \rangle$: by definition of popping elements from an empty list of resources.
- $\langle \mathbf{delete} \mid r_n \cdot s \mid l \rangle \rightsquigarrow \langle () \mid s \mid r_n :: l \rangle$: by definition of pushing an element in a list of resources.

E.4 Proof that centre 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 $E \in \mathscr{C}$ (see proposition 15):

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

Recall that a morphism $\mathscr{C}(A, B)$ is a function family $\prod_l A(l) \to B(l)$.
(2 $\implies$ 3) Resource-free objects have a trivial destructor:
We define the family of functions $d : \prod_l E(l) \to I(l)$ 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 $\forall l, E(l) \to I(l)$. If $l = []$, $I(l) = \emptyset$ so it must be the case that $E(l) = \emptyset$.

(1 $\implies$ 2) Objects in the Drinfeld centre have no resources. Such objects are pairs $E : \mathscr{C}, \theta : E \otimes - \cong - \otimes E$) such that

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

So, we have a bijection family

$$\prod_A \prod_l \prod_{j +\!\!+ k = l}, \sum_{j' +\!\!+ k' = l} E(j) \times A(k) \cong A(j') \times E(k') \ .$$

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 $j' +\!\!+ k' = r :: t +\!\!+ [r']$, there is no function $E(r :: t) \times A([r']) \to A(j') \times E(k')$: 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 centre. Given E resource-free, we define $\theta : \prod_A \prod_l \prod_{j+\!\!+k=l} \sum_{j'+\!\!+k'=l} E(j) \times A(k) \cong A(j') \times E(k')$ by case on j:

- If $j = []$ then $k = l$, so we pick $j' = k, k' = []$ and the function
$$\text{swap}_{E([]),A(k)} : E([]) \times A(k) \cong A(k) \times E([])$$
.
- 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 centre 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

$$\prod_l \prod_{i+\!\!+j+\!\!+k=l} \sum_{i'+\!\!+j'+\!\!+k'=l} E(i) \times A(j) \times B(k) \cong A(i') \times B(j') \times E(k')$$

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

$$f \otimes g : \prod_l \prod_{j+\!\!+k=l} \sum_{j'+\!\!+k'=l} A(j) \times C(k) \to B(j') \times D(k')$$

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

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