Information and Computation 301 (2024) 105234

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

Non-linear communication via graded modal session types

Danielle Marshall [a,b,*], Dominic Orchard [a,c]

[a] University of Kent, United Kingdom
[b] University of Glasgow, United Kingdom
[c] University of Cambridge, United Kingdom

ARTICLE INFO

Article history:
Received 29 November 2022
Received in revised form 1 October 2024
Accepted 29 October 2024
Available online 4 November 2024

Keywords:
Session types
Graded types
Communication patterns

ABSTRACT

Session types provide guarantees about concurrent behaviour and can be understood through their correspondence with linear logic, with propositions as sessions and proofs as processes. However, strict linearity is limiting since there exist useful communication patterns that rely on non-linear behaviours. For example, shared channels can repeatedly spawn a process with binary communication along a fresh linear channel. Non-linearity can be introduced in a controlled way through the concept of *graded modal types*, which are a framework encompassing various kinds of *coeffect* (describing how computations make demands on their context). This paper shows how graded modalities can work alongside session types, enabling various non-linear concurrency behaviours to be re-introduced precisely. The ideas are demonstrated using Granule, a functional language with linear, indexed, and graded modal types. We define a core calculus capturing the requisite features and our new graded primitives, then present an operational model and establish various key properties.

1. Introduction

Most programming languages ascribe a notion of *type* (dynamic or static) to data, classifying *what* data we are working with—integer, string, function, etc. Languages like Go and Rust among many others extend this to the realm of communication between two participants via *typed channels*, where any process with access to a channel endpoint can send and receive values of a given type across it. These typed channels only give us information about the nature of the data being communicated, however; to go beyond this we must consider *behavioural types*, which capture not just *what* data is, but *how* it is calculated.

One such behavioural type system for concurrency is *session types* [1,2], adding more structure to typed channels by incorporating a protocol that describes the precise behaviour of the processes communicating over the channel. As an example of such a protocol, consider the standard example (with some added flavour) of a customer who wishes to purchase a cake from a bakery. In natural language, we could describe the protocol as follows: the customer names their desired cake (a string), the bakery provides the price (an integer), and then the customer decides if they are happy with said price. If so, they confirm delivery details (another string), but if not, they end the communication immediately. From the customer's perspective, this protocol can be described as a session type, which would look something like:

* Corresponding author.
E-mail addresses: Danielle.Marshall@glasgow.ac.uk (D. Marshall), D.A.Orchard@kent.ac.uk (D. Orchard).

https://doi.org/10.1016/j.ic.2024.105234

```
Send String (Recv Int (Select (Send String End) End))
```

The particular constructs at play here will be explained in more depth later, but it is clear to see that this allows more information about the communication to be encoded in the type than simply the kinds of data that are being sent and received. We will also see later how the session type for the bakery can be understood as the *dual* to that of the customer, which helps to enforce type safety in the context of communication.

Session types naturally fit into the more general substructural discipline of *linear types* [3–5]. By treating data as a *resource* which must be used exactly once, linear types can capture various kinds of stateful protocols of interaction. Session types are inherently substructural: channels cannot in general be arbitrarily duplicated or discarded if they are to be used according to a precise sequence of operations (the protocol). This idea has allowed logical foundations to be developed for session types based on Curry-Howard correspondences with both intuitionistic [6,7] and classical linear logic [8,9]. Furthermore, linearly-typed functional languages then provide an excellent basis for session-typed programming [10]; session types can be represented in the linear lambda calculus via an additional interface for channels without any other changes to the type system [10,11].

Modern substructural type systems, however, allow us to go beyond the notion of linearity, classifying usage into more than just linear or non-linear. This idea originates from Bounded Linear Logic (BLL), which generalises the ! modality to a family of modalities indexed by a polynomial expressing an upper bound on usage [12]. For example, $!_{x \leq 2} A$ describes a proposition A which can be used at most twice.

From various directions this notion of BLL has been generalised further to develop *coeffect types* [13–15], which are a framework for capturing different analyses of a program's use of the context, all within a single type system. *Graded modal types* are an umbrella concept, encompassing coeffects and their dual notion of *effects*, and can provide an expressive system allowing for the specification and verification of many behavioural properties of programs [16]. In this paper, we demonstrate that combining session types with graded modal types (particularly for capturing coeffects) allows for various well-known non-linear behaviours of concurrent programs to be reintroduced in a linear setting in a controlled and precise way via the facilities of graded modalities.

A key part of this work is to reconcile the tension between three competing requirements: (1) side effects inherent in communication primitives, (2) non-linearity, and (3) the call-by-value semantics one might expect in most programming languages (in contrast to call-by-name, which is the basis of most theoretical explanations of linear and graded type systems). Requirements (1) and (2) can be satisfied more easily in a call-by-name setting, but (3) (CBV semantics) introduces unsoundness. Whilst a call-by-name semantics for a modal treatment of non-linearity captures the idea of a number of 'copies' of a value, a call-by-value semantics unifies these copies into just one value. In the context of concurrency, then, a non-linear channel in a CBV semantics points to a single channel and so multiple uses within a program can lead to deadlock or race-conditions. Our solution revolves around preventing resource allocating primitives (those that create channels) from being promoted to non-linear contexts, and representing multiplicity of channels through data structures. Section 4 describes in more detail the problem and our solution.

Granule is a research programming language built upon the core of the linear λ-calculus, extended with *graded modalities* as described above. A previous iteration of session types for Granule was described by Orchard et al. [16]. However, the system described required a monadic interface to avoid some issues caused by CBV; we solve these problems here to provide a more general and powerful interface for session-typed programming, and also demonstrate the broad range of possible applications for this interface by introducing a suite of primitives which capture the non-linear behaviours of reuse, replication, and repetition (providing *multicasting*).

We show three common patterns and how they can be described by combining linear session types and graded modal types: *reusable channels* (Section 6), *replicated servers* (Section 7), and *multicast communication* (Section 8). All of these patterns are a variation on the "fork" primitive, with some amount of substructural behaviour introduced via graded modal types, and sometimes restrictions to protocols via *type predicates*. All of the ideas presented here are implemented in the latest versions of the Granule compiler; interested readers who wish to experiment for themselves should feel free to download and install the most recent release.[1] We develop the operational semantics of our approach, via a core calculus (Section 5), and establish that type safety is preserved (which encompasses communication safety) in the presence of the new non-linear concurrency primitives (Section 9).

Section 10 discusses related work and Section 11 discusses further work, including a sketch of how to prove deadlock freedom, and various other potential extensions and applications.

Changelog The present paper is an extended version of work appearing at the PLACES 2022 workshop, with the more voluminous title of *Replicate, Reuse, Repeat: Capturing Non-Linear Communication via Session Types and Graded Modal Types* [17]. The present paper greatly expands the technical development, introducing a formal calculus and its operational semantics for both the core language and the concurrency primitives introduced here. In this version, we prove that session types combined with graded modal types satisfy syntactic type safety. Furthermore, we have expanded the number of primitives considered, added more examples and fixed a number of technical errors from the original paper.

[1] The latest build of the Granule compiler is always available from https://github.com/granule-project/granule/releases.

2. Granule and graded modal types primer

Granule's type system is based on the linear λ-calculus augmented with *graded modal types* [16] as well as polymorphic and indexed types (in the form of user-definable GADTs). With linear typing as the basis, we cannot write functions that discard or duplicate their inputs as in a standard functional programming language. However, we can introduce non-linearity via graded modalities and use these to represent such functions, exemplified by the following Granule functions:

```
drop : ∀ {a : Type} . a [0] → ()
drop [x] = ()

copy : ∀ {a : Type} . a [2] → (a, a)
copy [x] = (x, x)
```

The function arrow can be read as the type of linear functions (which consume their input exactly once; note that these are generally notated —∘, but in Granule functions are linear by default, so we simply use →), but `a [r]` is a graded modal type capturing the capability to use the value 'inside' in a non-linear way as described by `r`, i.e., `drop` uses the value 0 times and `copy` uses it 2 times. The pattern match `[x]` on the left-hand side of each function eliminates the graded modality, binding `x` as a non-linear variable. The Haskell-style tuple syntax `(a, a)` used here represents a linear tensor product type, which we later notate using the ⊗ operator in our core calculus.

The central idea of graded modal types is to capture program structure via an indexed family of modalities where the indices have some algebraic structure which gives an abstract view of program structure. We focus on *semiring graded necessity* in this paper (written $\Box_r A$ in mathematical notation but `A [r]` in Granule) which generalises linear logic's ! [3] and Bounded Linear Logic [12]. The semiring structure is also equipped with a preorder. The above example uses the natural number graded modality with equality as its preorder, thus providing an analysis which counts exactly how many times a value can be used.

Another useful graded modality has grades drawn from a semiring of *intervals* which allows us to give bounds on how a value might be used. To demonstrate, we define the classic `fromMaybe` function which allows a value that may or may not exist to be retrieved from a `Maybe` type.

```
data Maybe a = Just a | Nothing

fromMaybe : ∀ {a : Type} . a [0..1] → Maybe a → a
fromMaybe [_] (Just x) = x;
fromMaybe [d] Nothing  = d
```

Note that without the graded modality, this function would be ill-typed in Granule, since values are linear by default and one of the cases requires discarding the default value (given by the first parameter). By giving this parameter a type of `a [0..1]`, we specify that it can be used *either* `0` times or `1` time (in other words, this value has *affine* behaviour rather than linear). There is only one total function in Granule which inhabits the type we give to `fromMaybe` here–linearity forbids defining an instance which always returns the default value.

In order to use `fromMaybe` we need to 'promote' its first input to be a graded modal value, which is written by wrapping a value in brackets, e.g., `fromMaybe [42]`. Promotion propagates any requirements implied by the graded modality to the free variables. For example, the following takes an input and shares the capabilities implied by its grade to two uses of `fromMaybe`:

```
fromMaybeIntPlus : Int [0..2] → Maybe Int → Maybe Int → Int
fromMaybeIntPlus [d] x y = fromMaybe [d] x + fromMaybe [d] y
```

The `0..1` usage implied by the first `fromMaybe [d]` and `0..1` usage by the second are added together to get the requirement that the incoming integer `d` is graded as `0..2`.

Lastly, Granule includes *indexed* types which offer a lightweight form of dependency, allowing for type-level access to information about data. For example, length-indexed vectors can be defined and used as follows:

```
data Vec (n : Nat) (a : Type) where
  Nil  : Vec 0 a;
  Cons : a → Vec n a → Vec (n + 1) a

append : ∀ {a : Type, n m : Nat} . Vec n a → Vec m a → Vec (n + m) a
append Nil ys         = ys;
append (Cons x xs) ys = Cons x (append xs ys)
```

Indexed types allow us to ensure at the type-level that when we append two vectors, the length of the output is equal to the sum of the lengths of the two inputs. Again, thanks to linearity, we gain further assurances from our type signatures–here we can guarantee that since we must use every element of the input vectors, these must also appear in the output, and so no information is being lost along the way.

Indexed natural numbers can be similarly defined to vectors, and are used later to provide lightweight type-dependence on natural number inputs:

```
data N (n : Nat) where
    Z : N 0;
    S : ∀ {n : Nat} . N n → N (n+1)
```

This definition gives a way to capture a runtime witness for a type level `Nat`.

Granule allows type indices to be used also as grades, thus we can define for example the functorial action on vectors which links the size of the vector with the usage of the function to be applied to each element:

```
vmap : ∀ {a b : Type, n : Nat} . (a → b) [n] → Vec n a → Vec n b
vmap [_] Nil         = Nil;
vmap [f] (Cons x xs) = Cons (f x) (vmap [f] xs)
```

In Granule, predicates on types can be represented similarly to Haskell's type class constraints with type signatures of the form:

```
functionName : ∀{typeVariables} . {constraints} ⇒ type
```

For example, we can define a much more general version of `copy` as:

```
copy : ∀ {a : Type, s : Semiring, r : s}
     . {((1 + 1) : s) ⩽ r} ⇒ a [r] → (a, a)
copy [x] = (x, x)
```

This is polymorphic in the semiring `s`, as well as including a constraint as a predicate that the grade `r` is approximated by $1+1$ in this particular semiring. Types appearing in this constraint position have kind `Predicate`. We later introduce a number of type functions, returning a type of kind `Predicate`, in order to restrict protocols in various settings.

3. Core calculus, and linear session-typed channels

In order to later establish key properties of our approach, we replay the details of Granule's core calculus from the work of Orchard et al. [16], which gives a linear type theory with graded modal types. We extend this core calculus with our session-typed concurrency primitives. We outline the syntax and static semantics (type system) in this section. Section 5 defines an operational semantics that incorporates the notion of processes and channels which was not previously given for Granule's core calculus.

The core type theory we will focus on extends the linear λ-calculus with products, a multiplicative unit, and a *semiring-graded necessity modality* $\Box_r A$, where for a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ there exists a family of types $\{\Box_r A\}_{r \in \mathcal{R}}$. Being a pre-ordered semiring requires that $+$ and $*$ must be monotone with respect to $\sqsubseteq$ (but no other requirements beyond the semiring axioms). This language represents a simplified monomorphic subset of Granule [16], but closely resembles other graded systems from the literature [18–22]; the session type system presented in this paper should also be compatible with these, to varying extents depending on which kinds of grading the system in question accommodates.

Syntax Our syntax is that of the linear λ-calculus with multiplicative products and unit (first line of syntax below) with additional terms for introducing and eliminating the $\Box$ modality (second line):

$$\begin{aligned} t ::= {} & x \mid \lambda x.t \mid t_1\, t_2 \mid (t_1, t_2) \mid \mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2 \mid \mathbf{unit} \mid \mathbf{let}\ \mathbf{unit} = t_1\ \mathbf{in}\ t_2 \\ & \mid [t] \mid \mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2 \end{aligned} \tag{terms}$$

Following the syntax of variables, we group terms above into pairs of introduction and elimination forms, for functions, tensors, units, and the graded modal respectively. The meaning of these terms is explained in the next subsection with reference to their typing.

3.1. Type system

Typing judgements are of the form $\Gamma \vdash t : A$, assigning type A to a term t under context Γ. Types A are defined:

$$A, B ::= A \multimap B \mid A \otimes B \mid \mathbf{unit} \mid \Box_r A \tag{types}$$

Hence, our type syntax comprises linear function types $A \multimap B$, linear multiplicative products $A \otimes B$, a linear multiplicative unit **unit**, and the graded modality $\Box_r A$.

Contexts Γ contain both linear assumptions $x : A$ and graded assumptions $x : [A]_r$ which have originated from inside a graded modality. Syntax and typing are then given by the following rules:

$$\frac{}{x : A \vdash x : A}\,\text{VAR} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\multimap_i \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\multimap_e$$

The VAR, $\multimap_i$, and $\multimap_e$ rules are the standard rules of the linear λ-calculus, augmented with a notion of *contraction* captured by the $+$ operation on contexts coming from multiple sub-terms. This operation is only defined when contexts are disjoint with respect to linear assumptions; on overlapping graded assumptions we add their grades, e.g. $(\Gamma_1, x : [A]_r) + (\Gamma_2, x : [A]_s) = (\Gamma_1 + \Gamma_2), x : [A]_{r+s}$. More explicitly, context addition is specified as follows:

$$\begin{array}{rl} (\Gamma, x : A) + \Gamma' = (\Gamma + \Gamma'), x : A & \text{iff } x \notin |\Gamma'| \\ \Gamma + (\Gamma', x : A) = (\Gamma + \Gamma'), x : A & \text{iff } x \notin |\Gamma| \\ (\Gamma, x : [A]_r) + (\Gamma', x : [A]_s) = (\Gamma + \Gamma'), x : [A]_{(r+s)} & \end{array} \qquad \begin{array}{l} \emptyset + \Gamma = \Gamma \\ \Gamma + \emptyset = \Gamma \end{array}$$

Note that we treat contexts as unordered, so this specification is sufficient for cases where graded assumptions for the same variable appear in different positions within the two contexts.

The rules involving graded modalities are perhaps more interesting:

$$\frac{\Gamma \vdash t : A}{r * \Gamma \vdash [t] : \Box_r A}\,\text{PR} \qquad \frac{\Gamma_1 \vdash t_1 : \Box_r A \quad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let } [x] = t_1 \textbf{ in } t_2 : B}\,\text{ELIM}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\,\text{DER} \qquad \frac{\Gamma \vdash t : B}{\Gamma, 0 * \Gamma' \vdash t : B}\,\text{WEAK} \qquad \frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \quad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\,\text{APPROX}$$

The PR rule (promotion) introduces a graded modality with grade r and thus must scale by r all of the input assumptions Γ, where scaling is a partial operation defined:

$$r * \emptyset = \emptyset \qquad r * (\Gamma, x : [A]_s) = (r * \Gamma), x : [A]_{r*s}$$

Thus, $r * \Gamma$ is undefined if Γ contains any linear assumptions and therefore implicitly in the PR rule we require that Γ contains only graded assumptions. We later denote a context which contains only graded variables as $[\Gamma]$.

The ELIM rule eliminates a graded modality and captures the idea that a requirement for x to be used in an r-like way in t_2 can be matched with the capability of t_1 described by its graded modal type. In Granule, this construct is folded into pattern matching (seen above): we can 'unpack' (eliminate) a graded modality via pattern matching to provide a non-linear (graded) variable in the body of the function (the analogue to t_2 in this rule).

The DER rule ('dereliction') connects linear typing to graded typing, showing that a requirement for a linear assumption is satisfied by an assumption graded by 1. Note that it is not the case that $A \cong [A]_1$ since dereliction only converts from a graded assumption to linear assumption; there is no inverse. Meanwhile, WEAK explains how we can *weaken* with variables graded by 0. The last important rule is the APPROX rule which provides grade approximation; this allows a grade r to be converted to another grade s, provided that s *approximates* r where $\sqsubseteq$ is the pre-order of the semiring in question.

Example 1. Some useful choices for the pre-ordered semiring (all of which are provided in Granule) include:

- The semiring of natural numbers, which we define as $(\mathbb{N}, \times, 1, +, 0, \equiv)$. Note that the *discrete ordering* $\equiv$ here means that this particular semiring has no approximation, and so we are tracking the exact number of times a term has been used. We could instead use the standard $\leq$ ordering on natural numbers which would then permit approximation; this would allow for an *upper bound* on a term's usage.
- A semiring of intervals over $\mathbb{N}$ [16] (seen earlier), with the standard interval arithmetic definitions for $+$ and $*$, captures lower- and upper-bound usage, with $(l_1..u_1) \sqsubseteq (l_2..u_2)$ if $l_2 \leq l_1$ and $u_1 \leq u_2$.
- Intervals over the extended natural number semiring, with $\mathbb{N} \cup \{\infty\}$; these are additionally useful since constructing an interval with bounds $0..\infty$ allows us to capture a potentially arbitrary amount of use.
- The 'none-one-tons' semiring [23] with $\mathcal{R} = \{0, 1, \omega\}$ (in Granule, these are written `Zero`, `One`, `Many`) for tracking linear vs. non-linear usage, but with zero use identified as separate to non-linearity to capture irrelevance.
- The semiring of security levels [24], with $\mathcal{R} = \{\mathsf{Lo}, \mathsf{Hi}\}$ such that $\mathsf{Lo} \sqsubseteq \mathsf{Hi}$ (in the ordering that we use in this paper) which means that high-security inputs cannot flow to low-security outputs, and $1 = \mathsf{Lo}$ and $0 = \mathsf{Hi}$ with $* = \sqcap$ and $+ = \sqcup$. This can be generalised to ordered lattices with an arbitrary number of security levels, as are often used for tracking confidentiality and related properties in type systems which focus on information flow control [25].

Finally, we have the rules for introducing and eliminating tensor products and the multiplicative unit, which are standard, though it is of course important to remember that products are *linear* and so it is not possible to discard either side: both elements must be used when consuming a product.

$$\frac{\Gamma_1 \vdash t_1 : A \quad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\otimes_i \qquad \frac{\Gamma_1 \vdash t_1 : A \otimes A' \quad \Gamma_2, x : A, y : A' \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ (x, y) = t_1\ \textbf{in}\ t_2 : B}\otimes_e$$

$$\frac{}{\emptyset \vdash \textbf{unit} : 1}\text{UNIT}_i \qquad \frac{\Gamma_1 \vdash t_1 : 1 \quad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let unit} = t_1\ \textbf{in}\ t_2 : B}\text{UNIT}_e$$

The following gives an example that combines together several of the above rules to show how a value of type $\Box_3 A$ presents the capability to use an A value three times, which is split into forming a pair with a single usage of A and the formation of a further graded modality:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}\text{VAR}}{x : [A]_1 \vdash x : A}\text{DER} \quad \dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}\text{VAR}}{x : [A]_1 \vdash x : A}\text{DER}}{x : [A]_2 \vdash [x] : \Box_2 A}\text{PR}}{x : [A]_3 \vdash (x, [x]) : A \otimes \Box_2 A}\otimes_i \quad y : \Box_3 A \vdash y : \Box_3 A}{y : \Box_3 A \vdash \textbf{let}\ [x]\ = y\ \textbf{in}\ (x, [x]) : A \otimes \Box_2 A}\text{ELIM}$$

Note that $\Box_r A$ has the structure of a *graded exponential comonad* [24] meaning it has the following operations, all of which can be derived from the typing rules here: $\delta : \Box_{(r*s)} A \multimap \Box_r(\Box_s A)$ and $\varepsilon : \Box_1 A \multimap A$, and $\mathsf{contr} : \Box_{(r+s)} A \multimap (\Box_r A) \otimes (\Box_s A)$ and $\mathsf{weak} : \Box_0 A \multimap 1$.

3.2. Linear session types in granule

The implementation of session types in Granule is based on the GV calculus (originating from Gay and Vasconcelos [10], further developed by Wadler [8], for which we use Lindley and Morris' formulation [11]). The GV system extends the linear λ-calculus with a data type of channels $\mathsf{Chan}(S)$ parameterised by session types S [2], which capture the protocol of interaction allowed over the channel. Communication in GV is usually asynchronous: sending always succeeds, but receiving can block as usual, though Lindley and Morris consider a synchronous semantics [11].

Starting with a simple subset, session types can describe channels which send or receive a value of type T and then can be used according to session type S written $!T.S$ or $?T.S$ respectively, or can be closed written $\mathsf{end}_!$ or $\mathsf{end}_?$. There are then functions for sending or receiving a value on a channel, forking a process at one end of a channel returning the other, and waiting for a channel to be closed:

$$\begin{array}{ll} \mathsf{send} : T \otimes \mathsf{Chan}(!T.S) \multimap \mathsf{Chan}(S) & \mathsf{fork} : (\mathsf{Chan}(S) \multimap \mathsf{Chan}(\mathsf{end}_!)) \multimap \mathsf{Chan}(\overline{S}) \\ \mathsf{recv} : \mathsf{Chan}(?T.S) \multimap T \otimes \mathsf{Chan}(S) & \mathsf{wait} : (\mathsf{Chan}(\mathsf{end}_?) \multimap 1) \end{array}$$

where $\overline{S}$ is the *dual* session type to S, defined $\overline{!T.S} = ?T.\overline{S}$, $\overline{?T.S} = !T.\overline{S}$, $\overline{\mathsf{end}_?} = \mathsf{end}_!$ and $\overline{\mathsf{end}_!} = \mathsf{end}_?$. The fork combinator leverages the duality operation, spawning a process which applies the parameter function with a fresh channel, thus returning the dual endpoint.

The core interface in Granule correspondingly has operations with the following types, where `Protocol` is the kind of session types whose constructors and associated operations we highlight in purple:

```
send        : ∀ {a : Type, p : Protocol} . LChan (Send a p) → a → LChan p
recv        : ∀ {a : Type, p : Protocol} . LChan (Recv a p) → (a, LChan p)
forkLinear  : ∀ {p : Protocol}           . (LChan p → ()) → LChan (Dual p)
close       : LChan End → ()
```

The type-level function `Dual : Protocol → Protocol` provides the standard notion of duality for session types, defined:

```
Dual End = End

Dual (Send a p) = Recv a (Dual p)

Dual (Recv a p) = Send a (Dual p)

Dual (Select p1 p2) = Offer (Dual p1) (Dual p2)

Dual (Offer p1 p2) = Select (Dual p1) (Dual p2)
```

We can define a parallel composition combinator on thunked computations (i.e., of type `() → a`) using `forkLinear` to run one computation in parallel and then return its result via the associated channel:

```
-- 'par' expects two thunks, 'f' and 'g'
-- and returns their resulting values in a pair
par : ∀ {a : Type, b : Type} . (() → a) → (() → b) → (a, b)
par f g =  let                    -- compute f in new thread
               c       = forkLinear (λc' → close (send c' (f ())));
               b       = g ();    -- compute g in current thread
               (a, c') = recv c; -- get result of f from other thread
               ()      = close c'
           in (a, b)
```

Choice We also provide some utility operations for internal and external choice (also known as selection and offering); note that as described in the original formulation by Lindley and Morris [11] these can be defined in terms of the core functions given above, and so when it comes to our semantics we will not need to define explicit reductions for or prove properties relating to these operations, but we do provide them as primitives in the implementation for ease of use.

```
selectLeft  : ∀ {p1 p2 : Protocol} . LChan (Select p1 p2) → LChan p1
selectRight : ∀ {p1 p2 : Protocol} . LChan (Select p1 p2) → LChan p2
offer       : ∀ {p1 p2 : Protocol, a : Type}
            . (LChan p1 → a) → (LChan p2 → a) → LChan (Offer p1 p2) → a
```

The following gives a brief example putting all these primitives together:

```
server : LChan (Offer (Recv Int End) (Recv () End)) → Int [0..1] → Int
server c [d] = offer (λc→ let (x, c) = recv c in let () = close c in x)
                     (λc→ let ((), c) = recv c in let () = close c in d) c

client : ∀ {p : Protocol} . LChan (Select (Send Int End) p) → ()
client c = let c = selectLeft c;
               c = send c 42
           in close c

example : Int -- Evaluates to 42
example = server (forkLinear client) [100]
```

The `server` offers a choice between being sent an integer or a unit value. The second parameter (bound to `d`) is used as a default value in the case that a unit value is received by the server, where `Int [0..1]` denotes that this integer can be used 0 or 1 times (see Section 2 for more explanation of this grading). The `client` selects the left behaviour, sends an integer, then closes its side of the communication.

The final definition `example` spawns the client with a channel and connects the dual end of the channel to the server which returns the received value of `42` here.

Core calculus representation In the core calculus we add a type of channels indexed by session types (protocols), with grammar:

$$A ::= \ldots \mid \mathbf{Chan}\, P \qquad \text{(types, extended)}$$

$$P ::= \mathbf{Send}\, A\, P \mid \mathbf{Recv}\, A\, P \mid \mathbf{End} \qquad \text{(session types)}$$

with the duality function $\overline{P}$ on session types defined as expected:

$$\overline{(\mathbf{Send}\, A\, P)} = \mathbf{Recv}\, A\, \overline{P} \quad \overline{(\mathbf{Recv}\, A\, P)} = \mathbf{Send}\, A\, \overline{P} \quad \overline{\mathbf{End}} = \mathbf{End}$$

As mentioned above, we elide the offer and select types from the core calculus and the theoretical development in this paper, since they can also be emulated using send and receive and coproducts as in the Fst presentation of GV [11].

The term language is extended with the following primitives and their typing:

$$\begin{aligned}
\mathsf{send} &: \mathbf{Chan}\,(\mathbf{Send}\, A\, P) \multimap A \multimap \mathbf{Chan}\, P \\
\mathsf{receive} &: \mathbf{Chan}\,(\mathbf{Recv}\, A\, P) \multimap A \otimes \mathbf{Chan}\, P \\
\mathsf{forkLinear} &: (\mathbf{Chan}\, P \multimap 1) \multimap \mathbf{Chan}\, \overline{P} \\
\mathsf{close} &: \mathbf{Chan}\, \mathbf{End} \multimap 1
\end{aligned}$$

We assume asynchronous communication, which is captured in the operational model of Section 5.

4. The relationship between grading, call-by-value, and effects

The type system described thus far is based heavily on that of Orchard et al. [16], though with some simplifications where we elide concepts that are irrelevant to this paper. Orchard et al. also described an earlier version of session types in Granule based somewhat on the GV calculus, but the session types were not included in the language's formal semantics at the time, which resulted in some cases where combining grading and session types led to unsoundness and the possible introduction of deadlocks. This work resolves these issues and we will present a full operational semantics in Section 5 as part of our formal theory of graded session types, but before we do this we first discuss the problems present in the original system and describe our solution.

The first and primary problem becomes apparent when promoting a primitive function which creates new channels. Consider the following example which is allowed by the type system described so far:

```
1  problematic : Int
2  problematic =
3    let [c] : ((LChan (Recv Int End)) [2]) = [forkLinear (λc → close (send c 42))];
4        (n, c') = recv c;  () = close c';
5        (m, c') = recv c;  () = close c'
6    in (n + m)
```

On line 3, the program forks a process that sends 42 on a channel, but under a promotion, with the type explaining that we want to use the resulting value twice (given by the explicit type signature here). This promotion then allows two uses of the channel on lines 4-5.

The typical semantics for coeffect-based calculi in the literature is call-by-name [13,24,15,19]. Under a call-by-name semantics, which Granule allows via the extension `language CBN`, this program executes and produces the expected result of `84`. The key is that call-by-name reduction substitutes the call to `forkLinear` into the two uses of variable `c` on lines 4 and 5, and thus we are receiving from two different channels. However, under the default call-by-value semantics (which was chosen in Granule for simplicity, performance, and to avoid complications resulting from effects), line 3 is fully evaluated (underneath the graded modal introduction), and so variable `c` on lines 4 and 5 refers to a single channel. This means that executing this program blocks indefinitely on line 5; we get an error from the underlying implementation's concurrency primitives (written in Haskell) describing a `thread blocked indefinitely in an MVar operation`. This comes from Granule's runtime which leverages these standard primitives.

In order to get around this problem with non-linear channels in a call-by-value setting, Orchard et al. [16] instead deal with channels monadically, so here we would end up with a channel of type `(Session (LChan ...)) [2]` (with the `Session` monad).[2] To make this example well-defined would then require a distributive law which maps from this to `Session ((LChan ... ) [2])`, copying the channel. Providing such a distributive law is unsound—it would enable `problematic` and thus indefinite blocking—and fortunately it is also *not* derivable. However, we wish to avoid the monadic programming style as it is not required when working with just linear channels.

Our alternative solution is that we instead syntactically restrict promotion to terms which do not allocate resources, i.e., whose reduction does not create linear channels. In particular this includes reducible expressions containing the `forkLinear` primitive which causes the difficulty here. Non-linear channels are then re-introduced by additional primitives in Sections 6, 7 and 8 which allow for precise and carefully managed non-linear usage, allowing for grading to be combined with linear channels even in a call-by-value setting and without the extra overhead that was required when every channel had to be wrapped in the `Session` monad.

Thus, we modify the promotion rule to be:

$$\frac{\Gamma \vdash t : A \qquad \neg \mathsf{resourceAllocator}(t)}{r * \Gamma \vdash [t] : \Box_r A}\ \mathrm{PR}$$

where $\mathsf{resourceAllocator}(t)$ classifies those terms whose reduction to a value would trigger creation of a resource; such terms contain a use of the forkLinear primitive (or the other fork primitives introduced in later sections) in a reducible position. Note in particular that $\neg\mathsf{resourceAllocator}(\lambda x.\mathsf{forkLinear}\, t)$ since reduction does not happen underneath a λ-abstraction (Section 5 defines the reduction semantics). Definition 6 in Appendix A gives the formal definition. Generally, any term whose reduction semantics creates channels is classified as a resource allocator.

Promotion is not the only source of unsoundness in the session type system of Orchard et al., [16] however. Some additional restrictions on the primitives which combine grading with linear channels are required, in order to prevent possible problems that can occur when two ends of the same channel reach inconsistent communication states or are used at different grades. We will discuss these problems further in Sections 6, 7 and 8 where we will go on to define how exactly these new primitives must operate in order to properly preserve communication safety.

[2] In Granule such a channel would in fact have the type `(LChan ... <Session>) [2]`, with `<Session>` being an instance of the graded monadic `<r>` modalities which are dual to the graded comonadic `[r]` modalities. We elide discussing these further here since we will not need any monad other than `Session`.

Linear Haskell Note that since we developed this approach, other work has resolved the same problem relating to promotion through different techniques. Originally, the linear types extension to Haskell [20] avoided this difficulty by only ever allocating new resources (such as linear channels or mutable arrays) inside a continuation, which is passed around at every step until deallocation to prevent the allocator itself from ever being promoted.

More recently, this strategy of requiring specialised restrictions on the interface for each resource allocator has been generalised by introducing a "linear constraint" [26] called `Linearly`. This constraint is assumed whenever a new resource is allocated, and since the assumption must be used exactly once this prevents the promotion of a linear resource constructor in much the same way. A continuation is still necessary when `Linearly` is initially assumed, but unlike before this same qualification can now be used generically across many types of resource.

5. Semantics

We define an operational model for our linear session types here, in order to make the connection between our type system and the underlying communication behaviours more concrete, and so that we can prove that type safety will continue to hold after our extensions with additional primitives in Sections 6, 7, and 8. We give a semantics to terms in the context of runtime "configurations" of processes and channels. This allows us to show that progress can always be made regardless of the current state of the global configuration, and proving preservation of typing of the overall configuration. We discuss deadlock freedom as further work in Section 11, with a sketch of the key theorems and definitions.

5.1. Operational semantics

The semantics is call-by-value and at its core is based on the standard small-step operational semantics for the linear λ-calculus and its extension to graded modal types in a call-by-value setting given by Orchard et al. [16]. We extend this with the processes and channels.

Processes A *process configuration* tracks the state of a number of processes, comprising ordinary terms, running in parallel:

$$\mathcal{P} ::= \cdot \mid (\mathcal{P} \mid t^{\mathcal{R}}) \qquad \text{(process configurations)}$$

$$\mathcal{R} ::= \circ \mid \bullet \qquad \text{(role)}$$

(the parentheses are typically omitted). The role $\circ$ marks a subordinate process and $\bullet$ marks the main process. Typing of process configurations later establishes that a well-typed process configuration has exactly one main process (Section 5.2).

Configurations are subject to an equivalence relation capturing commutativity and associativity of the parallel composition in process configurations:

$$\mathcal{P}_1 \mid \mathcal{P}_2 \equiv \mathcal{P}_2 \mid \mathcal{P}_1 \qquad (\mathcal{P}_1 \mid \mathcal{P}_2) \mid \mathcal{P}_3 \equiv \mathcal{P}_1 \mid (\mathcal{P}_2 \mid \mathcal{P}_3)$$

In subsequent relations (e.g., reduction) we work up-to process equivalence, i.e., we can freely apply the equivalence relation to rearrange process configurations. This relation is akin to *structural congruence* in process algebras.

Channels Communication channels in the operational semantics are represented by pairs of dual *channel endpoints* to allow asynchronous duplex communication. Throughout, c ranges over channel endpoint names, which appear in 'positive' and 'negative' forms:

$$c ::= n^+ \mid n^- \qquad \text{(channel endpoint names)}$$

The *dual* of an endpoint name c is another endpoint name, denoted $\overline{c}$, where $\overline{n^+} = n^-$ and $\overline{n^-} = n^+$ and thus duality is an involution, i.e., $\overline{\overline{c}} \equiv c$.

Channel configurations record the contents of channel endpoints in the current environment by associating names with a queue $\vec{v}$ of message values:

$$\mathcal{C} ::= \cdot \mid \mathcal{C}, c = \vec{v} \qquad \text{(channel configurations)}$$

We write $\langle\rangle$ for an empty queue, $\vec{v} : v$ for a queue with value v enqueued on the end, and $v : \vec{v}$ for a queue with v on the front (ready to be dequeued).

We will sometimes refer to c as simply a channel, and also as a reference to its contents (the queue) within a configuration.

Reduction Single-step global reductions are then defined over pairs of process and channel configurations (which we refer to as *global configurations*) with reductions of the form:

$$(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')$$

where $\mathcal{P}, \mathcal{P}'$ are (incoming and outgoing) process configurations and $\mathcal{C}, \mathcal{C}'$ are (incoming and outgoing) channel configurations respectively. The channel configuration may be updated if reduction involves applying a concurrency primitive, which could involve spawning two endpoints of a new channel or communicating a value from one end of a channel to the other.

Channels inside a channel configuration can either be *linear* or they can be *shared*, which means that rather than being used linearly they are used according to some grade r. We will discuss this in more detail when we describe primitives that can spawn graded channels, starting from Section 6.

The standard call-by-value small-step semantics for the linear λ-calculus embeds within this more complex global configuration semantics through the use of the following rule:

$$\frac{t \rightsquigarrow t'}{(\mathcal{P} \mid t^{\mathcal{R}};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid t'^{\mathcal{R}};\ \mathcal{C})}\ \textsc{reduce}$$

which states that if the term t can reduce to t' (without any need to interact with the global context) then we can make that same reduction within the global configuration, at any role, whilst threading all of the extra channel configuration information through so that it can come back into play once we reach a communication step. Going forwards, we elide the role in places where a reduction rule is agnostic to the role, assuming that the role of a process being reduced is preserved in its corresponding location in the output process configuration.

We will now briefly discuss how these standard reduction steps work in practice, to provide a foundation from which we can go on to explain the behaviour of our core session type primitives in this setting of process and channel configurations. We will explain some rules in detail and elide others which are very similar to those already introduced. All reduction rules for the entire semantics are collected together in Appendix A.

The core reduction rules for the (CBV) linear λ-calculus terms are of course β-reduction and function application, which are given as follows.

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\rightsquigarrow_{\text{APP}} \qquad \frac{t \rightsquigarrow t'}{v\ t \rightsquigarrow v\ t'}\rightsquigarrow_{\text{prim}} \qquad \frac{}{(\lambda x.t)\ v \rightsquigarrow t[v/x]}\rightsquigarrow_{\beta}$$

Our semantics is call-by-value, and left-to-right: we fully reduce the left-hand side of a function application to a value and then reduce on the right.

Perhaps more interesting are the rules for "boxed" terms which are those typed with a graded modality in our system; note that in many other Granule-related papers with call-by-need semantics any boxed term is a value by default, but here a boxed term is only a value if the term inside the box is itself a value. The reduction rules are:

$$\frac{}{\textbf{let}\ [x]\ = [v]\ \textbf{in}\ t \rightsquigarrow t[v/x]}\rightsquigarrow_{\Box\beta} \qquad \frac{t \rightsquigarrow t'}{[t] \rightsquigarrow [t']}\rightsquigarrow_{\Box\text{reduce}}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let}\ [x]\ = t_1\ \textbf{in}\ t_2 \rightsquigarrow \textbf{let}\ [x]\ = t_1'\ \textbf{in}\ t_2}\rightsquigarrow_{\text{LET}\Box}$$

Thus we can β-reduce an introduction-elimination pair for the graded modal values, eliminating the box modality through pattern matching. We reduce inside the box for terms that are not yet values, and we have a congruence rule for reducing the term in an elimination form. We will skip over the rules for tensor product and unit in this section, since they are very similar to the rules for boxed terms, with rules for reducing terms inside a pair and also substitution and elimination for products and unit as above.

The real action happens when we reach an instance of application where the value being applied on the left is some (potentially already partially applied) communication primitive and the term on the right is the final parameter for said primitive, as this is where some communication behaviour needs to occur and so we will have to consider the state of the channel configuration. We begin by considering the creation of a new channel, which occurs through the use of the forkLinear primitive. For this we need to now extend the syntax of terms to include *runtime values*, which in this setting are just channel endpoint names c:

$$t ::= \ldots \mid c \qquad \text{(Runtime terms)}$$

Furthermore, we also define the notion of an evaluation context, denoted *Ctxt*, in which reduction can occur at a single "hole" in the syntax tree:

Definition 1 *(Evaluation contexts).* Syntax trees with a single hole are defined:

$$
\begin{array}{lll}
Ctxt & ::= & \text{Contexts} \\
& | & - \\
& | & Ctxt\ t \\
& | & v\ Ctxt \\
& | & [Ctxt] \\
& | & \mathbf{let}\,[x] = Ctxt\,\mathbf{in}\,t \\
& | & \mathbf{let}\,\mathbf{unit} = Ctxt\,\mathbf{in}\,t \\
& | & \mathbf{let}\,(x, y) = Ctxt\,\mathbf{in}\,t \\
& | & (Ctxt, t) \\
& | & (v, Ctxt)
\end{array}
$$

where $-$ denotes a hole and we write $Ctxt\{t\}$ to denote the operation of filling the hole in $Ctxt$ with a term t.

The operational semantics of forkLinear, in some evaluation context, is then:

$$\frac{\#c}{(\mathcal{P} \mid Ctxt\{\mathsf{forkLinear}\,(\lambda x.t)\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^\circ \mid Ctxt\{\overline{c}\};\ \mathcal{C}, \overline{c} = \langle\rangle, c = \langle\rangle)}\ \textsc{ForkLinear}$$

In this case, we select a fresh name c for the new channel (denoted by $\#c$ in the premise) and add both dual endpoints of the channel (c and $\overline{c}$) to the channel configuration, each empty of values for now. As well as reducing the term by carrying out the expected substitution we also spawn a new process for communication on the fresh channel and add it to the process configuration, marked as a subordinate process.

In the forkLinear rule, we need not concern ourselves with whether the fresh endpoint name c is a positive channel name n^+ or negative n^- since the actual polarity is immaterial; whatever fresh name we have, the channel configuration is extended with a queue associated to this endpoint name c and a queue associated to its dual endpoint name $\overline{c}$.

The next cases we consider are sending and receiving information on one end of a channel, where the dual end will have to receive or provide said information in turn. Sending is asynchronous, given by:

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{send}\,c\,v\};\ \mathcal{C}, \overline{c} = \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{c\};\ \mathcal{C}, \overline{c} = \vec{v} : v)}\ \textsc{Send}$$

The value v provided by the user is consumed, and the channel configuration is updated by appending v to the end of whatever queue of values is already present inside the *dual* end of the channel they sent on, where they will later be necessarily received by virtue of the session type guarantees. The process configuration contains the original end of the channel in context, which the sender will continue to use according to the next step of its protocol given by the session type. Conversely, receiving operates as follows:

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{receive}\,c\};\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{(v, c)\};\ \mathcal{C}, c = \vec{v})}\ \textsc{Recv}$$

In this case, the first value is popped from the queue of values contained in the end of the channel held by the receiver, and this is given to the user inside the context in the process configuration; they can again continue to use their end of the channel according to its protocol. This may involve receiving more values or sending values to the dual end depending on the particular session type that has been specified.

Finally, channels are closed as follows:

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{close}\,c\};\ \mathcal{C}, c = \langle\rangle) \rightsquigarrow (\mathcal{P} \mid Ctxt\{\mathbf{unit}\};\ \mathcal{C})}\ \textsc{Close}$$

Note this reduction closes the referred-to endpoint of the channel c, rather than its dual, akin to how RECV receives from the referred to endpoint of the channel. This allows for asynchronous behaviour since the dual endpoint $\overline{c}$ may still have values in its queue to be received.

5.2. Runtime typing

In order to later consider type safety properties, we extend typing to process and channel configurations, considering a notion of *runtime typing*.

For runtime typing of terms, the same typing rules apply but where the context also contains channel names and their typing. We refer to a *runtime context* by the letter Δ, defined:

$$\Delta = \emptyset \mid \Delta, c : \mathbf{Chan}\,P \qquad \text{(Runtime contexts)}$$

$$\Gamma = \ldots \mid \Delta \qquad \text{(Contexts (redefined))}$$

where $c : \mathbf{Chan}\, P$ types a linear channel. Contexts Γ are then extended to include Δ, though it is sometimes useful to denote contexts that contain only runtime types (channel types) separately by using the Δ notation.

Context addition extends to Δ, treating the linear channel names just as linear variables. The variable rule of typing extends to channel names, i.e., we can derive runtime typing:

$$\frac{}{c : \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\,\text{VAR}$$

Process configurations $\mathcal{P}$ are typed by a judgment $\Gamma, \Delta \vdash \mathcal{P} : A$ where A is the type of the main process and Γ are the term variables and Δ the channel names in scope of the processes in $\mathcal{P}$. Here, the type $\bot$ is a type that cannot be given to any term, but is used only for process configuration typing to denote a configuration without a main process.

$$\frac{}{\emptyset \vdash \cdot : \bot}\,\text{EMPTY}$$

$$\frac{\Gamma_1, \Delta_1 \vdash \mathcal{P} : \bot \qquad \Gamma_2, \Delta_2 \vdash t : A}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash \mathcal{P} \mid t^{\bullet} : A}\,\text{MAIN} \qquad \frac{\Gamma_1, \Delta_1 \vdash \mathcal{P} : A \qquad \Gamma_2, \Delta_2 \vdash t : 1}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash \mathcal{P} \mid t^{\circ} : A}\,\text{SUB}$$

Empty process configurations are given the type $\bot$. The MAIN rule puts a term t in parallel with a configuration $\mathcal{P}$ and marks it as the main process and thus the process configuration has the type of this process. The SUB rule puts a term t in parallel with a configuration $\mathcal{P}$, but this term is marked as a subordinate process with $\circ$ (of which there may be many) and thus the return type of t must be **unit**. The process configuration's type is that of $\mathcal{P}$. Note that this type can itself be $\bot$, and so a process configuration with type $\bot$ is not necessarily empty.

Channel configurations $\mathcal{C}$ are typed by a judgment $\Delta \vdash \mathcal{C}$, defined:

$$\frac{}{\emptyset \vdash \cdot}\,\text{NIL}$$

$$\frac{\Delta \vdash \mathcal{C}}{\Delta, c : \mathbf{Chan}\, P \vdash \mathcal{C}, c = \langle\rangle}\,\text{EMPTY} \qquad \frac{\Delta \vdash \mathcal{C}}{\Delta, c : \mathbf{Chan}\, \mathbf{End} \vdash \mathcal{C}, \overline{c} = \vec{v}}\,\text{END}$$

$$\frac{\Delta_1 \vdash v : A \qquad \Delta_1, \Delta_2, c : \mathbf{Chan}\, P \vdash \mathcal{C}, \overline{c} = \vec{v} : v}{\Delta_1, \Delta_2, c : \mathbf{Chan}\, (\mathbf{Send}\, A\, P) \vdash \mathcal{C}, \overline{c} = \vec{v}}\,\text{SEND}$$

$$\frac{\Delta_1 \vdash v : A \qquad \Delta_1, \Delta_2, c : \mathbf{Chan}\, P \vdash \mathcal{C}, c = \vec{v}}{\Delta_1, \Delta_2, c : \mathbf{Chan}\, (\mathbf{Recv}\, A\, P) \vdash \mathcal{C}, c = v : \vec{v}}\,\text{RECV}$$

The EMPTY rule says that a channel endpoint that is empty can have any type (it is in a 'waiting state'). Meanwhile, the END rule says a channel can be typed at **Chan End** while its dual endpoint may contain any number of values. Reading the SEND rule bottom-up, the conclusion types a channel on which a value v of type A can be sent and thus we can type that channel with the continuation behaviour P in the premise when that value has been enqueued on the channel. Note that channel value is the dual to the typing, since sending is actioned by enqueueing on the dual endpoint. Reading the RECV rule bottom-up, the conclusion types a channel on which a value v of type A can now be received (from the head of the channel) with the continuation behaviour P in the premise after the value has been received (and hence removed) from the front of the channel.

5.2.1. Summary

So far we have established just the operational semantics for the core calculus with linear channels. We are now in a position to establish the key properties of well-typed programs containing graded modal session types but we delay this to Section 9.

Next, we introduce the new non-linear communication primitives which are the main contribution of this paper and expand the operational semantics to include their model, and associated definitions, which also includes expanding the definition of processes to include some intermediate runtime processes. In Section 9, we then consider the syntactic type safety properties for the full calculus with the non-linear extensions included.

6. Reusable channels

A reusable or *non-linear* channel is one which can be shared and thus used repeatedly in a sound way. This contrasts with the notion seen in Section 4 of promoting a fresh linear channel to being non-linear in a call-by-value setting, which was unsound; a linear channel used in a shared way can easily lead to a deadlock where one user of the channel leaves it in a state which is then blocking for another user of the channel. The key problem with sharing a channel is ending up with inconsistent states across different shared usages. This can be avoided if the protocol allowed on the channel is restricted

such that only a single 'action' (send, receive, choice, or offer) is allowed, and thus if the channel is used multiple times it can never be left in an inconsistent state across shared uses. This is captured by the idea that $(P \mid P) \neq P$ in general in process calculi (like CCS or the π-calculus) and so a replicated channel which has more than a single action cannot be split off into many parallel uses, e.g., $(ab)^* \mid (ab)^* \not\equiv (ab)^*$. However, if there is only a single action then multiple repeated parallel uses are consistent, e.g., $a^* \mid a^* \equiv a^*$.

We capture this idea via the following variant of the fork primitive with arbitrarily graded channels:

```
forkNonLinear : ∀ {p : Protocol, s : Semiring, r : s}
              . {SingleAction p, ExactSemiring s}
             ⇒ ((LChan p) [r] → ()) → (LChan (Dual p)) [r]
```

where `ExactSemiring : Semiring → Predicate` restricts acceptable semirings to only those that do not permit any approximation (so as far as semirings mentioned in this paper are concerned, the grade `r` is allowed to be a natural number, representing exact usage, but importantly is *not* allowed to be an interval, representing a range of possible usages) and `SingleAction : Protocol → Predicate` is a type constraint that characterises only those protocols which comprise a single send, receive, choice, or offer, i.e.,

```
SingleAction End      SingleAction (Send a End)      SingleAction (Offer End End)
                      SingleAction (Recv a End)      SingleAction (Select End End)
```

As an example, consider a channel in a graded modality which says it can be used exactly `n` times. The following uses this channel to send every element of a vector of size `n` (using the `Vec` type of Section 2):

```
1  sendVec : ∀ {n : Nat, a : Type} . (LChan (Send a End)) [n] → Vec n a → ()
2  sendVec [c] Nil = ();
3  sendVec [c] (Cons x xs) =
4        let c'  = send c x;
5            ()  = close c'
6        in sendVec [c] xs
```

This code shows the powerful interaction between grading, linearity, and indexed types in Granule. Note that the above code is typeable without any of the new primitives described in this section, but would be unusable without promoting a use of `forkLinear`. However, we can complete this example with a dual process (`recvVec`) that is then connected to `sendVec` via `forkNonLinear`:

```
1  recvVec : ∀ {n : Nat, a : Type} . N n → (LChan (Recv a End)) [n] → Vec n a
2  recvVec Z [c]      = Nil;
3  recvVec (S n) [c] = let (x, c') = recv c;
4                          () = close c'
5                      in Cons x (recvVec n [c])
6
7  length' : ∀ {n : Nat, a : Type} . Vec n a → (N n, Vec n a)
8  length' Nil = (Z, Nil);
9  length' (Cons x xs) = let (n, xs) = length' xs in (S n, Cons x xs)
10
11 example : ∀ {n : Nat, a : Type} . Vec n a → Vec n a
12 example xs = let (n, list) = length' xs
13              in recvVec n (forkNonLinear (λc→ sendVec c list))
14
15 main : Vec 5 Int
16 main = example (Cons 1 (Cons 1 (Cons 2 (Cons 3 (Cons 5 Nil)))))
```

Since `forkNonLinear` gives us a pair of channels whose semiring-usage is exact, we must consume all values produced by `sendVec` on the dual endpoint. Therefore, the receiver `recvVec` needs to know how many elements to receive. This information is passed separately (via an indexed natural number `N n`) so that `recvVec` can induct on the natural number and leave no message unreceived. A system with dependent session types [27,7] could avoid this by first sending the length, but this is not (yet) possible in Granule; the Gerty prototype language provides full dependent types and graded modal types which would be a good starting point [28].

6.1. Semantics

In the notation of the core calculus, the `forkNonLinear` primitive has its typing rendered as:

$$\frac{\mathsf{SingleAction}(P) \qquad \mathsf{ExactSemiring}(S) \qquad r : S}{\emptyset \vdash \mathsf{forkNonLinear} : (\Box_r \mathbf{Chan}\, P \multimap 1) \multimap \Box_r(\mathbf{Chan}\, \overline{P})}\ \textsc{forkNonLin}$$

This primitive now introduces the notion of a non-linear, or *shared* channel, which we reflect into the runtime typing and channel configuration typing that was setup in Section 5.

As an example (partial) derivation here, consider the following term which uses forkNonLinear with a protocol **Send Int End**, and the natural numbers semiring with grade 2 to spawn a process reusing the channel twice to send two integer terms (3 then 5), yielding the dual channel which can be used twice to receive two integers:

$$\dfrac{\dfrac{\dfrac{y : [\mathbf{Chan}\,(\mathbf{Send}\,\mathbf{Int}\,\mathbf{End})]_2 \vdash \mathbf{let}\ \mathbf{unit} = \mathsf{close}\,(\mathsf{send}\,y\,3)\ \mathbf{in}\ \mathsf{close}\,(\mathsf{send}\,y\,5) : 1}{x : \square_2(\mathbf{Chan}\,(\mathbf{Send}\,\mathbf{Int}\,\mathbf{End})) \vdash \mathbf{let}\ [y]\ = x\ \mathbf{in}\ \mathbf{let}\ \mathbf{unit} = \mathsf{close}\,(\mathsf{send}\,y\,3)\ \mathbf{in}\ \mathsf{close}\,(\mathsf{send}\,y\,5) : 1}\ \textsc{elim}}{\emptyset \vdash \lambda x.\mathbf{let}\ [y]\ = x\ \mathbf{in}\ \mathbf{let}\ \mathbf{unit} = \mathsf{close}\,(\mathsf{send}\,y\,3)\ \mathbf{in}\ \mathsf{close}\,(\mathsf{send}\,y\,5) : 1}\ \multimap_i}{\emptyset \vdash \mathsf{forkNonLinear}\,(\lambda x.\mathbf{let}\ [y]\ = x\ \mathbf{in}\ \mathbf{let}\ \mathbf{unit} = \mathsf{close}\,(\mathsf{send}\,y\,3)\ \mathbf{in}\ \mathsf{close}\,(\mathsf{send}\,y\,5)) : \square_2(\mathbf{Chan}\,(\mathbf{Recv}\,\mathbf{Int}\,\mathbf{End}))}\ \textsc{forkNonLin}$$

Shared channel typing We extend runtime-typing contexts Δ with a shared channel typing:

$$\Delta = \cdot \mid \Delta, c : \mathbf{Chan}\,P \mid \Delta, c :^* \mathbf{Chan}\,P \qquad \text{(Runtime contexts)}$$

where $c :^* \mathbf{Chan}\,P$ types a shared channel.

The predicate $[\Gamma]$ classifying graded contexts then also extends to include the rule that $[\Gamma] \Longrightarrow [\Gamma, c :^* \mathbf{Chan}\,P]$, i.e., we treat shared channels akin to graded variables, as graded variables may be used non-linearly. Runtime context addition then needs to account for sharing:

Definition 2 *(Runtime context addition).* Runtime context addition is akin to context addition, where channels can be marked as linear or shared and addition is only defined for contexts that are disjoint in their linear channels and where shared channels may appear in either context:

$$\begin{aligned}
\emptyset + \Delta_2 &= \Delta_2 \\
\Delta_1 + \emptyset &= \Delta_1 \\
(\Delta_1, c : \mathbf{Chan}\,P) + \Delta_2 &= (\Delta_1 + \Delta_2), c : \mathbf{Chan}\,P \quad \text{iff } c \notin \mathsf{dom}(\Delta_2) \\
\Delta_1 + (\Delta_2, c : \mathbf{Chan}\,P) &= (\Delta_1 + \Delta_2), c : \mathbf{Chan}\,P \quad \text{iff } c \notin \mathsf{dom}(\Delta_1) \\
(\Delta_1, c :^* \mathbf{Chan}\,P_1) + (\Delta_2, c :^* \mathbf{Chan}\,P_2) &= (\Delta_1 + \Delta_2), c :^* \mathbf{Chan}\,P \quad \text{where } P = P_1 \bowtie P_2
\end{aligned}$$

where $P_1 \bowtie P_2$ defines *compatibility of shared channel session types*, a symmetric partial operation defined:

$$\begin{aligned}
P \bowtie P &= P \quad \textit{if } \mathsf{SingleAction}(P) \\
P \bowtie \mathbf{End} &= P \quad \textit{if } \mathsf{SingleAction}(P) \\
\mathbf{End} \bowtie P &= P \quad \textit{if } \mathsf{SingleAction}(P)
\end{aligned}$$

where SingleAction is a built-in predicate on protocols:

$$\mathsf{SingleAction}(\mathbf{End}) \quad \mathsf{SingleAction}(\mathbf{Recv}\,A\,\mathbf{End}) \quad \mathsf{SingleAction}(\mathbf{Send}\,A\,\mathbf{End})$$

Thus, two shared channels are compatible when they have the same protocol, or when one of the shared channels has been used and is in the **End** state. For example, consider three non-linear uses of a shared channel c whose protocol is a single action $\mathbf{Send}\,A\,\mathbf{End}$; after one communication is performed, one of the uses of c is typed as **End**, but this is still compatible with the two remaining non-linear uses of c typed at $\mathbf{Send}\,A\,\mathbf{End}$.

Thus, $+$ combines contexts which are disjoint in their linear channels, and allows shared channels to be included in both contexts as long as their session types are compatible, computing via $\bowtie$ their compatible shared type.

Scalar multiplication of contexts extends smoothly to scalar multiplication of runtime contexts where $r * (\Delta, c :^* \mathbf{Chan}\,P) = (r * \Delta), c :^* \mathbf{Chan}\,P$.

The variable rule extends to shared channel names:

$$\dfrac{}{c :^* \mathbf{Chan}\,P \vdash c : \mathbf{Chan}\,P}\ \textsc{varChanS}$$

The weakening rule also extends to shared channel names:

$$\dfrac{\Gamma \vdash t : B}{\Gamma, 0 * \Gamma', c :^* \mathbf{Chan}\,P \vdash t : B}\ \textsc{weakChanS}$$

Channel configurations and reduction In channel configurations, we explicitly denote shared channels with assignments written $c^* = \vec{v}$. If we need to distinguish a linear channel in a channel configuration, we write $c^1 = \vec{v}$ though if the linear or shared nature is immaterial we write $c = \vec{v}$ as before. The typing of channel configurations behaves the same for shared channels as linear channels.

Send and receive reduction rules apply to shared channels exactly as linear channels. However, the close primitive has a different reduction semantics for shared channels:

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{close}\, c\};\ \mathcal{C}, c^* = \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{\mathbf{unit}\};\ \mathcal{C}, c^* = \vec{v})}\ \textsc{SharedClose}$$

Note that when a shared channel is closed, we preserve it inside the channel configuration rather than removing it. Furthermore, we do not even require that it is empty since the channel may be shared and messages may have arrived for another interaction, e.g., particularly in the case of forkNonLinear.

An important point here is that although the channel is preserved, unlike in the linear close rule, this still does not permit orphan messages to remain inside any channel when it is closed. This is because of the `ExactSemiring` constraint on the forkNonLinear primitive, which ensures that there are always exactly the same number of senders as receivers, and the `SingleAction` constraint, which restricts each participant to only send or receive a single message. Indeed, this is exactly why both of these restrictions are necessary.

Reduction semantics for `forkNonLinear` Now that we have extended the runtime typing notions to shared channels we can consider the operational semantics of forkNonLinear.

The operational model of forkNonLinear follows the typing: its reduct starts with an application of this primitive to a function value $\mathsf{forkNonLinear}\,(\lambda x.t)$ in some evaluation context within a process reducing by creating a fresh shared channel endpoint c (and its dual endpoint $\overline{c}$) and spawning a new process t with x replaced by the graded modal lifting of the channel $[c]$ and returning the dual endpoint also inside a graded modal box:

$$\frac{\#c}{\begin{array}{l}(\mathcal{P} \mid Ctxt\{\mathsf{forkNonLinear}\,(\lambda x.t)\};\ \mathcal{C}) \\ \rightsquigarrow\ (\mathcal{P} \mid Ctxt\{[\overline{c}]\} \mid t[[c]/x]^\circ;\ \mathcal{C}, \overline{c}^* = \langle\rangle, c^* = \langle\rangle)\end{array}}\ \textsc{ForkNonLinear}$$

The spawned process based on the function body t (with the promoted channel substituted in) is marked as being a subordinate process with $\circ$. The newly created channels in the context are marked as being shared channels c^* and $\overline{c}^*$.

7. Replicated servers

The π-calculus provides replication of a process P as the process $!P$, which session-typed π-calculus variants have refined into the more controlled idea of having a replicated 'server'. Here, the spawning of replicated instances is controlled via a special receive [29–31], e.g., written like $*c(x).P$ meaning receive an x on channel c and then continue as P, whilst still providing the original process. From an operational semantics point of view this looks like $*c(x).P \mid \overline{c}\langle d\rangle.Q \rightarrow *c(x).P \mid P[d/x] \mid Q$ where $\overline{c}\langle d\rangle.Q$ sends the message d to the server which 'spawns' off a fresh copy of the server process P whilst also preserving the original server process for further clients to interact with.

This important pattern is not achievable with the reusable channels of Section 6 but the non-linearity inherent to replicated servers can be captured and explained again via the combination of graded and indexed types. We provide the replicated server functionality here as the fork variant `forkReplicate`:

```
forkReplicate : ∀ {p : Protocol, n : Nat} . {ReceivePrefix p}
              ⇒ (LChan p → ()) [0..n] → N n → Vec n ((LChan (Dual p)) [0..1])
```

Here the grading is less general than in Section 6; we instead focus on a particular grading which says that given a server process `(LChan p → ())` that can be used 0 to n times, then we get a vector of size n of dual channels which we can use to interact with the server. The predicate `ReceivePrefix p` classifies those protocols which start with a receive, i.e.:

```
ReceivePrefix (Recv a p)      ReceivePrefix (Offer p1 p2)
```

Each returned client channel can itself be discarded due to the graded modality `... [0..1]`. Thus, we can choose not to use any/all of the client channels, reflected in the dual side where the server can be used at most n times. This introduces some flexibility in the amount of usage.

The following example demonstrates `forkReplicate` to define a simple server that can receive requests to receive two further integers which it adds and sends back, or requests to compare a received integer for equality to 0:

```
1  addServer : LChan (Offer (Recv Int (Recv Int (Send Int End)))
2                           (Recv Int (Send Bool End))) → ()
3  addServer c =
4   offer
5         (λc→ let (x, c) = recv c;
6                  (y, c) = recv c;
7               in close (send c (x + y))
8
9         (λc→ let (x, c) = recv c;
10              in close (send c (x == 0))) c
```

We then define two clients `client1` and `client2` that can interact with this server, exercising both parts of the offered choice:

```
client1 : ∀ {p : Protocol}
        . LChan (Select (Send Int (Send Int (Recv Int End))) p) → Int
client1 c = let (x, c) = recv (send (send (selectLeft c) 10) 20);
                () = close c
            in x

client2 : ∀ {p : Protocol}
        . LChan (Select p (Send Int (Recv Bool End))) → Bool
client2 c = let (b, c) = recv (send (selectRight c) 42);
                () = close c
            in b
```

We then compose the server and clients together using `forkReplicate` to get a vector of channels, which we pattern match on immediately to get the channels to pass to each client (which are composed in parallel using `par` defined in Section 3.2):

```
import Parallel -- Provides the 'par' combinator derived from 'forkLinear'
main : (Int, Bool)
main =
  let (Cons [c1] (Cons [c2] Nil)) = forkReplicate [addServer] (S (S Z))
  in par (λ () → client1 c1) (λ () → client2 c2)
```

In the intuitionistic linear logical propositions of Caires and Pfenning [6], this same idea of a replicating fork is captured by a non-linear channel (in Dual Intuitionistic Linear Logical style) which yields a linear version when interacted with; this is akin to the idea of *shared channels* [2]. Our approach codifies the same principle using graded modalities, but only allows finite replication. Section 7.2 shows a variation of this primitive which allows unbounded replication.

7.1. Semantics

In the notation of the core calculus, the `forkReplicate` primitive has its typing rendered as:

$$\frac{\mathsf{ReceivePrefix}(P)}{\emptyset \vdash \mathsf{forkReplicate} : \Box_{0..n}(\mathbf{Chan}\, P \multimap 1) \multimap \mathbf{N}\, n \multimap \mathbf{Vec}\, n\, (\Box_{0..1}(\mathbf{Chan}\, \overline{P}))}\ \textsc{forkRep}$$

We thus also extend the core calculus with indexed types for natural numbers and vectors, which previously were derived inside Granule using the syntax of GADTs (see Section 2):

$$A ::= \ldots \mid \mathbf{N}\, n \mid \mathbf{Vec}\, n\, A$$

These indexed types are internally definable in Granule as GADTs, but for simplicity we add them as primitive to the core calculus, with standard introduction rules:

$$\frac{}{\emptyset \vdash \mathbf{Z} : \mathbf{N}\, 0}\ \textsc{Z} \qquad \frac{\Gamma \vdash t : \mathbf{N}\, n}{\Gamma \vdash \mathbf{S}\, t : \mathbf{N}\, (n+1)}\ \textsc{S}$$

$$\frac{}{\emptyset \vdash \langle\rangle : \mathbf{Vec}\, 0\, A}\ \textsc{nil} \qquad \frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : \mathbf{Vec}\, n\, A}{\Gamma_1 + \Gamma_2 \vdash t_1 :: t_2 : \mathbf{Vec}\, (n+1)\, A}\ \textsc{cons}$$

We elide their eliminators here for brevity, but their elimination is standard following usual eliminators for indexed types [32] (see also Granule's type system [16], whose indexed type eliminators also cause specialisation in grades, allowing type indices and grades to interact). As a shorthand, we write a vector constructor with n elements as $\langle t_1, .., t_n \rangle$.

As an example (partial) derivation, the following uses forkReplicate to spawn a server that can be used at most three times to receive an integer and sends it back incremented:

$$\frac{\dfrac{\emptyset \vdash \lambda x.\mathbf{let}\ (y, x) = \mathsf{receive}\, x\ \mathbf{in}\ \mathsf{close}\, (\mathsf{send}\, x\, (y+1)) : \mathbf{Chan}\, (\mathbf{Recv}\, \mathbf{Int}\, (\mathbf{Send}\, \mathbf{Int}\, \mathbf{End})) \multimap 1}{\emptyset \vdash [\lambda x.\mathbf{let}\ (y, x) = \mathsf{receive}\, x\ \mathbf{in}\ \mathsf{close}\, (\mathsf{send}\, x\, (y+1))] : \Box_{0..3}(\mathbf{Chan}\, (\mathbf{Recv}\, \mathbf{Int}\, (\mathbf{Send}\, \mathbf{Int}\, \mathbf{End})) \multimap 1)}\ \textsc{pr}}{\emptyset \vdash \mathsf{forkReplicate}\, [\lambda x.\mathbf{let}\ (y, x) = \mathsf{receive}\, x\ \mathbf{in}\ \mathsf{close}\, (\mathsf{send}\, x\, (y+1))] : \mathbf{N}\, 3 \multimap \mathbf{Vec}\, 3\, (\Box_{0..1}(\mathbf{Chan}\, (\mathbf{Send}\, \mathbf{Int}\, (\mathbf{Recv}\, \mathbf{Int}\, \mathbf{End}))))}\ \textsc{forkRep}$$

The operational model of forkReplicate splits into two rules. The first follows the typing and corresponds to a β-reduction of forkReplicate: its reduct has an application of this primitive to a promoted function application and a natural number term $\mathsf{forkReplicate}\, [\lambda x.t]\, n$. The result is to spawn off a number of intermediate processes called "watchers" and to return to the originating process a vector of fresh channel endpoints (each promoted), of type $\mathbf{Vec}\, n\, (\Box_{0..1}(\mathbf{Chan}\, \overline{P}))$:

$$\frac{\#c_1 \ldots \#c_n}{(\mathcal{P} \mid \mathcal{C}txt\{\mathsf{forkReplicate}\, [\lambda x.t]\, n\};\ \mathcal{C})}\ \textsc{forkReplicate}$$

$$\rightsquigarrow\ (\mathcal{P} \mid \mathcal{C}txt\{\langle [\overline{c_1}], .., [\overline{c_n}] \rangle\} \mid \mathsf{Wch}\, (c_1 \mapsto \lambda x.t) \mid .. \mid \mathsf{Wch}\, (c_n \mapsto \lambda x.t)\ ;\ \mathcal{C}, {c_1}^* = \langle\rangle, .., {c_n}^* = \langle\rangle, \overline{c_1}^* = \langle\rangle, .., \overline{c_n}^* = \langle\rangle)$$

Thus, n pairs of fresh channel endpoints are created and the forkReplicate reduct, in some evaluation context $Ctxt$, is replaced by a vector of the dual endpoints (promoted to the graded modality). In parallel, n "watcher" processes are created, where $\mathsf{Wch}\,(c \mapsto t)$ is a runtime process waiting to receive a value on channel c and containing the continuation of the replicated process body: the suspended computation of process t, abstracted over the channel x.

Watcher processes (which are not ordinary terms) have the following typing, under the process typing judgment:

$$\frac{\Gamma_1, \Delta_1 \vdash \mathcal{P} : A \qquad \Gamma_2, x : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P), \Delta_2 \vdash t : 1}{\Gamma_1 + (0..1 * \Gamma_2), \Delta_1 + \Delta_2, c : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \vdash \mathcal{P} \mid \mathsf{Wch}\,(c \mapsto \lambda x.t) : A}\,\text{WATCH}$$

Thus, given a t which has a free x which is a channel on which we can receive, then t can be placed into a watcher process, linked to the channel c on which a receive can occur. Note that in the conclusion here we scale the requirement of t by $0..1$ since t may never be evaluated as forkReplicate provides an upper bound on the number of possible replications.

The second part of the operational semantics for forkReplicate then involves reducing a watcher process into a regular term-based process upon receiving an initial request:

$$\frac{}{(\mathcal{P} \mid \mathsf{Wch}\,(c \mapsto \lambda x.t);\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^{\circ};\ \mathcal{C}, c = v : \vec{v})}\,\text{WATCHER}$$

For a watcher on channel c, if there is a value v at the end head of the channel v then a new subordinate process is formed, substituting c for x. Note that the channel is left unchanged so that the spawned process can receive the value; the presence of the value v in the channel simply triggers the spawning of the process from the watcher.

7.2. Variations

In `forkReplicate`, each returned channel (in a vector) can be discarded due to the graded modality `... [0..1]`: we can choose not to use any/all of the client channels, which allows for some flexibility in the amount of usage. A more strict variant is given as:

```
forkReplicateExactly : ∀ {p : Protocol, n : Nat} . {ReceivePrefix p}
    ⇒ (LChan p → ()) [n] → N n → Vec n (LChan (Dual p))
```

meaning we have exactly n clients that *must* spawn the server n times.

A more powerful version of replication allows an unknown number of clients (possibly infinite), with an interface similar to `forkReplicate` but returning a lazy stream of client channels which can be affinely used:

```
data Stream a = Next a (Stream a)

forkReplicateForever : ∀ {p : Protocol} . {ReceivePrefix p}
    ⇒ (LChan p → ()) [0..∞] → (Stream (LChan (Dual p))) [0..1]
```

The result is a stream wrapped in the graded modality with grade `0..1`. Due to the typing of pattern matching in Granule, one can pattern match on the stream inside the graded modality, such that its components inherit the grade `0..1`; e.g., `example [Next c rest] = ...` binds the stream `rest` in the scope of the function body with the same grade `0..1` so it can be used once or not at all.

This last variation of replication captures more closely the replicated server concept found in session-typed π-calculi. The above two primitives are provided in the implementation but are not considered in the core calculus and metatheory here.

8. Multicast sending

The final primitive we introduce here provides the notion of *multicast* (or *broadcast*) communication where messages on a channel can be received by multiple processes by copying the messages. This non-linearity is thus on the payload values being sent, with a graded modal type deployed to explain that the amount of non-linearity in the messages matches the number of receiving processes (clients). The `forkMulticast` primitive provides this behaviour, typed in Granule:

```
forkMulticast : ∀ {p : Protocol, n : Nat}
              . {Sends p}
             ⇒ (Chan (Graded n p) → ()) → N n → Vec n (Chan (Dual p))
```

where `Graded : Nat → Protocol → Protocol` is a built-in type function that adds a graded modality to payload types, defined:

```
Graded n End            = End
Graded n (Send a p)     = Send (a [n]) (Graded n p)
Graded n (Recv a p)     = Recv (a [n]) (Graded n p)
Graded n (Select p1 p2) = Select (Graded n p1) (Graded n p2)
Graded n (Offer p1 p2)  = Offer (Graded n p1) (Graded n p2)
```

and `Sends : Protocol → Predicate` is a built-in predicate defined:

$$\frac{}{\texttt{Sends End}} \qquad \frac{\texttt{Sends p}}{\texttt{Sends (Send a p)}} \qquad \frac{\texttt{Sends p1} \quad \texttt{Sends p2}}{\texttt{Sends (Select p1 p2)}}$$

Thus, as long as we are sending values that are wrapped in the graded modality such that they can be used n times, we can then broadcast these to n client processes via the `Vec n (Chan (Dual p))` channels returned by `forkMulticast`.

For example, in the following we have a `broadcaster` that takes a channel which expects an integer to be sent which can be used 3 times.

```
broadcaster : LChan (Send (Int [3]) End) → ()
broadcaster c = close (send c [42])
```

We can then broadcast these results with `forkMulticast broadcaster (S (S (S Z)))` producing a 3-vector of channels. Below we aggregate the results from these three receiver channels by applying `aggregateRecv` to the vector, giving us the result `126`, i.e. `3 * 42`.

```
aggregateRecv : ∀ {n : Nat} . Vec n (LChan (Recv Int End)) → Int
aggregateRecv Nil          = 0;
aggregateRecv (Cons c cs) = let (x, c) = recv c;
                                () = close c
                            in x + aggregateRecv cs

main : Int -- Evaluates to 126
main = aggregateRecv (forkMulticast broadcaster (S (S (S Z))))
```

8.1. Semantics

In the notation of the core calculus, the `forkMulticast` primitive has its typing rendered as:

$$\frac{\mathsf{Sends}(P)}{\emptyset \vdash \mathsf{forkMulticast} : (\mathbf{Chan}\,(\mathsf{Graded}_n P) \multimap 1) \multimap \mathbf{N}\,n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})}\ \textsc{forkMulti}$$

Similarly to forkReplicate, the operational model of forkMulticast comprises two rules; a β-reduction that generate some intermediate processes, and a rule actioning those processes in response to communication. The β-reduction rule follows the typing; its reduct has an application of the primitive to a function value and a natural number term: $\mathsf{forkMulticast}\,(\lambda x.t)\,n$. The result is to spawn the process derived from t as well as a special "forwarding" process (explained below):

$$\frac{\#c \qquad \#c_1 \ldots \#c_n}{\begin{array}{l}(\mathcal{P} \mid \mathcal{C}txt\{\mathsf{forkMulticast}\,(\lambda x.t)\,n\};\ \mathcal{C}) \\ \rightsquigarrow\ (\mathcal{P} \mid \mathcal{C}txt\{\langle \overline{c_1}, \ldots, \overline{c_n}\rangle\} \mid t[c/x]^\circ \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n) \\ \qquad ;\ \mathcal{C}, c = \langle\rangle, \overline{c} = \langle\rangle, c_1 = \langle\rangle, .., c_n = \langle\rangle, \overline{c_1} = \langle\rangle, .., \overline{c_n} = \langle\rangle)\end{array}}\ \textsc{forkMulticast}$$

Thus, n pairs of fresh channel endpoints are created, $c_1 \ldots c_n$, and the forkMulticast reduct is replaced by a vector of the dual endpoints $\langle \overline{c_1}, .., \overline{c_n}\rangle$. In parallel, the function body t from the reduct has c substituted for x to form a subordinate process $t[c/x]^\circ$, and a "forwarding" process is created. The forwarding process $\mathsf{Fwd}\,(c \mapsto c_1, .., c_n)$ is a runtime process that receives a graded modal value from c, unboxes the value and forwards copies of this payload on the (dual end of) channels c_1 to c_n.

Forwarder processes are typed by the process typing judgment as:

$$\frac{\mathsf{Sends}(P) \qquad \Gamma, \Delta \vdash \mathcal{P} : A}{\Gamma, \Delta, c : \mathbf{Chan}\,(\mathsf{Graded}_n \overline{P}), c_i : \mathbf{Chan}\,P \vdash \mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n) : A}\ \textsc{fwd}$$

where $c_i : \mathbf{Chan}\,P$ is short hand for the typing $c_1 : \mathbf{Chan}\,P, .., c_n : \mathbf{Chan}\,P$. For example, if $P = \mathbf{Send}\,B\,P'$ (satisfying the Sends predicate) then $c : \mathbf{Chan}\,(\mathbf{Recv}\,(\Box_n B)\,(\mathsf{Graded}_n \overline{P'}))$ in the above.

The reduction of the forwarding process then occurs by receiving on channel c an n-graded value, which gets sent along n further channels:

$$\frac{}{\begin{array}{l}(\mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n);\ \mathcal{C}, c = [v] : \vec{v}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n) \\ \rightsquigarrow\ (\mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n);\ \mathcal{C}, c = \vec{v}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v)\end{array}}\ \textsc{forwarder}$$

Note that there is no "garbage collection" step for the forwarding process, though this can be easily added as an implementational detail.

9. Properties

We consider the standard probes on the language design: whether typing guarantees progress of reduction (that we do not get stuck, or blocked locally) and preservation of types (that reduction does not conflict with the specifications of types and maintains its meaning throughout). Proofs of all the statements that follow are provided in the appendix.

We are now in a position to establish the key properties of well-typed programs containing graded modal session types. Our semantics defined over global configurations that was presented in Section 5 allows us to demonstrate that type safety (and thus communication safety) holds by proving two main theorems: progress and preservation.

Lemma 1. *Value lemma*
Given $\Gamma \vdash v : A$ then, depending on the type, the shape of v can be inferred:

- $A = A' \multimap B$ *then* $v = \lambda x.v'$ *or a partially applied primitive term* p.
- $A = \Box_r A'$ *then* $v = [v']$.
- $A = A' \otimes B$ *then* $v = (v_1, v_2)$.
- $A = 1$ *then* $v = \mathbf{unit}$.
- $A = \mathbf{Chan}\, P$ *then* $v = c$.

Note that as in other session-typed systems, typability of an expression is not sufficient to guarantee that it can immediately make progress at runtime [10]. For example, a channel with type $\mathbf{Chan}\,(\mathbf{Recv}\,A\,P)$ can only make progress if the dual channel has already sent the value that this channel needs to receive (as described by the Recv rule). Our progress theorem applies to well-typed configurations that also satisfy the following property.

Definition 3. Given $\Delta \vdash t : A \ \wedge\ \Delta \vdash \mathcal{C}$ then we say that $\mathcal{C}$ is *buffered* with respect to Δ, written $\mathsf{buffered}(\mathcal{C}, \Delta)$, if for any channel c in $\mathcal{C}$ with type $\mathbf{Chan}\,(\mathbf{Recv}\,A\,P)$, we have that c contains at least one value v.

The full inductive definition of the buffered predicate is as follows, where $c :^r \ldots$ in the runtime contexts and $c =^r \ldots$ in the process configurations abstracts over both linear and shared channels.

$$\frac{}{\mathsf{buffered}(\cdot, \Delta)} \qquad \frac{\mathsf{buffered}(\mathcal{C}, \Delta)}{\mathsf{buffered}((\mathcal{C}, c =^r \vec{v}), (\Delta, c :^r \mathbf{Chan}\,\mathbf{End}))}$$

$$\frac{\mathsf{buffered}(\mathcal{C}, \Delta)}{\mathsf{buffered}((\mathcal{C}, c =^r \vec{v}), (\Delta, c :^r \mathbf{Chan}\,(\mathbf{Send}\,A\,P)))}$$

$$\frac{\mathsf{buffered}(\mathcal{C}, \Delta)}{\mathsf{buffered}((\mathcal{C}, c =^r v : \vec{v}), (\Delta, c :^r \mathbf{Chan}\,(\mathbf{Recv}\,A\,P)))}$$

Theorem 1 *(Progress terms).*

$$\begin{aligned} & \Delta \vdash t : A \\ \wedge\ & \Delta \vdash \mathcal{C} \\ \wedge\ & \mathsf{buffered}(\mathcal{C}, \Delta) \\ \wedge\ & \Delta \vdash \mathcal{P} : B \\ \Longrightarrow\ & \mathsf{value}(t) \vee \exists t', \mathcal{C}', \mathcal{P}'.\ (\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t';\ \mathcal{C}') \end{aligned}$$

A process configuration $\mathcal{P}$ contains all values, defined by the predicate $\mathsf{values}(\mathcal{P})$, if every process is either a value v or is a forwarder or watcher process.

Theorem 2 *(Progress global config).*

$$\begin{aligned} & \Delta \vdash \mathcal{P} : A \ \wedge\ \Delta \vdash \mathcal{C} \ \wedge\ \mathsf{buffered}(\mathcal{C}, \Delta) \\ \Longrightarrow\ & \mathsf{values}(\mathcal{P}) \vee \exists \mathcal{C}', \mathcal{P}'.(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}') \end{aligned}$$

Proof. (Highlights) See Appendix C for the full proof.

Note that there are two separate progress theorems here; the first is akin to the standard progress theorem on terms, though some cases require additional information from the process and channel configurations and the overall result is that

either the term t is a value, or a global step can be taken which can reduce either or both of the two configurations along with the term. The second global progress theorem consider cases where we must reduce a process configuration without being able to focus on a term, for example if we need to reduce based on watcher and forwarder processes which only exist at runtime.

Most cases in the first progress theorem proceed exactly as they would in a standard proof of progress for a type system; in these cases the Reduce rule from the global semantics allows us to say that if a term can be reduced in the standard way, then we can take a global step which does not affect the remainder of the process configuration or any of the channels which exist at runtime, simply reducing that term in place.

The interesting cases in this proof are the ones where a communication primitive is applied to a value, which by the value lemma (Lemma 1) must be a channel. In these cases we cannot reduce by only considering the term; the rules we must use are the global semantic rules for the communication primitives, which take into account the channels that are present in the channel configuration and in some cases (such as ForkReplicate and ForkMulticast) also spawn additional processes.

The second proof is not too involved, and almost follows as a corollary of the first, but does require a little extra work for the cases of watcher and forwarder processes. Note in particular that it is possible for a forwarder process to still be present and for no reduction to be possible, but this is only in cases where all other terms are values, so the theorem does still hold here because $\mathsf{values}(\mathcal{P})$ holds for a process configuration as long as all terms have been fully reduced. □

Theorem 3 *(Preservation).*

$$\begin{aligned} &\Gamma, \Delta \vdash \mathcal{P} : A \ \wedge\ \Delta \vdash \mathcal{C} \ \wedge\ (\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}') \\ \Longrightarrow\ &\exists \Delta'.\ \Gamma, \Delta' \vdash \mathcal{P}' : A \ \wedge\ \Delta' \vdash \mathcal{C}' \end{aligned}$$

Proof. (Highlights) See Appendix D for the full proof.

The proof proceeds by induction on the structure of process configuration typing, in concert with channel configuration typing and then the inductive definition of global reductions that are possible at each typing.

Preservation for reduction of a single term comes from previous preservation results for Granule's core calculus [16].

For the concurrent primitives (both the communication primitives and forking primitives introduced here), their reductions involve terms in a reduction context. A lemma provides *factorisation* for a well-typed term in context, i.e., if $Ctxt\{t\}$ is well-typed then $Ctxt\{x\}$ is well-typed and t is well-typed separately (Lemma 6, p. 28). This lemma tells us that either x is a linear variable, i.e., the term t is substituted into a linear hole position, or x is a graded variable, i.e., the term t is substituted into a graded hole position, which for $\Gamma \vdash Ctxt\{t\} : B$ then gives us:

$$\begin{aligned} (\exists \Gamma_1, \Gamma_2, A, r.\ &\neg \mathsf{resourceAllocator}(t) \\ &\wedge\ \Gamma_1, x : [A]_r \vdash Ctxt\{x\} : B \\ &\wedge\ [\Gamma_2] \vdash t : A \\ &\wedge\ \Gamma = \Gamma_1 + r * \Gamma_2) \end{aligned}$$

Critical here is that we know that this factorisation tells us that t is not a resource allocator. Given a global reduction which leads to a new term t', we then have the evidence to show that $\neg\mathsf{resourceAllocator}(t')$ in these cases. *Context application* lemmas then give the dual result allowing us to reconstruct a well-typed term in context, i.e., $Ctxt\{t'\}$ is well-typed (with preserved types).

Note that communication primitives (send, receive, close) are not resource allocators and can occur in graded contexts, in which case the factorisation lemma also states that their channels must therefore be shared, since $[\Gamma_2]$ classifies (runtime) contexts which are all graded or all shared channels. In these cases, the typing preservation is straightforward.

For the forking primitives, and auxiliary processes like forwarding and watching, the factorisation lemma tells us that these must necessarily be within a linear context as they are resource allocators, and thus given $\Gamma \vdash Ctxt\{t\} : B$ we can factorise the typing into $\Gamma_1, x : A \vdash Ctxt\{x\} : B$ and $\Gamma_2 \vdash t : A$ where $\Gamma = \Gamma_1 + \Gamma_2$. This shows us the effect of restricting terms to non-resource allocators in ensuring preservation. The type preservation then follows by the constructing the typing of the subject produced by reduction, which all satisfy the theorem statement.

For the forkReplicate case, the $\mathsf{ReceivePrefix}(P)$ predicate is essential to type preservation: the watcher processes that forkReplicate reduces into rely on having receiving channels in order to continue reducing. Furthermore, the typing of a forkReplicate reduct has its first argument of type $\square_{0..n}(\mathbf{Chan}\, P \multimap 1)$. Since this argument is a value in the reduct, by inversion (the value Lemma 1), it is necessarily of the form $[\lambda x.t]$ and we have $\Gamma_1, x : \mathbf{Chan}\, P, \Delta_1 \vdash t : 1$ and $0..n * (\Gamma_1, \Delta_1) \vdash [\lambda x.t] : (\mathbf{Chan}\, P \multimap 1)$ where Γ_1, Δ_1 is a graded context. Reduction yields n watchers, each of which holds a copy of t that by the typing of watchers promotes their resources by $0..1$ to $0..1 * (\Gamma_1, \Delta_1)$. Thus the n-wise parallel composition therefore (by distributivity of $*$ over $+$) results in preserving the $0..n * (\Gamma_1, \Delta_1)$ resources.

For the forkMulticast case, the $\mathsf{Sends}(P)$ predicate is essential to type preservation also: the forwarding processes that forkMulticast reduce into rely on having a receiving channel endpoint and subsequent channels to send upon. Further-

more, the payload grading given by **Graded** $n\,P$ ensures that the copying of the value to n channel endpoints is well-resourced. □

Progress and preservation go some way to ensuring that our approach does not undermine guarantees of the language. A further useful property to prove would be *deadlock freedom*. We give a sketch of how to prove this for our system in Section 11, although completing it for the full system here is left as further work.

10. Related work

Shared channels Session type theories often provide non-linearity via *shared names* which provide repeated behaviours such as a replicable 'server' [2] akin to our `forkReplicate` concept.

Particularly relevant in our context of linear types, Caires and Pfenning give a session-types meaning to intuitionistic linear logic in which propositions denote the types of (linear) channels and the exponential modality ! types the communication of shared servers which can be repeatedly spawned [6]; that is, $!A$ denotes the type of a non-linear shared channel which is used by a 'server' process to spawn new sessions conforming to the communication behaviour of A. The production of a new 'client' channel involves giving a semantic meaning to *dereliction* where a non-linear (shared) channel has a linear copy 'sharded' off. In their formulation, $\Gamma; \Delta \vdash P :: T$ types a process P providing behaviour T along a channel, and using non-linear channels Γ and linear channels Δ. Access to the replicable server then involves the following process and typing:

$$\frac{\Gamma, u : A; \Delta, y : A \vdash P :: T}{\Gamma, u : A; \Delta \vdash (\nu y)u\langle y\rangle.P :: T}$$

i.e., access to the shared server via shared channel $u : A$ (essentially a $!A$ linear channel) is provided to P by the new linear channel $y : A$, amounting to contraction of u and dereliction of u into y. This is coherent with a call-by-name semantics for linear logic where a value of type $!A$ is essentially an infinite number of copies of an A value, whereas in the call-by-value semantics $!A$ is just a single A value that can be used non-linearly. Therefore in our CBV approach, `forkReplicate` provides a vector of client channels to capture the notion of channel copies.

In the GV calculus, Lindley and Morris add replicated channels by first extending GV with an S4-style □ modality (akin to the exponential modality) to provide "unlimited types" [11]. They derive a replicate primitive of the following type (where in GV, the type $!A.S$ denotes a channel on which we first send an A then behave as S, see Section 3.2):

$$\mathsf{replicate} : (!(\Box\overline{S}).\mathsf{end}_!) \otimes (\Box(S \multimap \mathsf{end}_!)) \multimap \mathsf{end}_!$$

$$\mathsf{replicate}(x, f) = \mathsf{send}\,([\mathbf{let}\,[g] = f\,\mathbf{in}\,\mathsf{fork}\,f], x)$$

(we have used our notation for modal introduction and elimination to make the derivation more easily understood here).

Thus, replication takes a pair of (1) a channel x on which is sent a $\Box\overline{S}$ value, i.e., a non-linear channel which can behave as $\overline{S}$ and (2) a non-linear function f which expects a channel behaving as S as input). The function unboxes the non-linear function f into an instance g, and forks this, but underneath modal promotion, yielding a 'boxed' channel which is then sent on the input channel x. Whilst GV is call-by-value like Granule, the derivation above is only sound if □ behaves in a call-by-name way, i.e., we do not reduce under □ but instead capture the enclosing fork such that uses of it on the dual endpoint of x get a freshly forked function g each time. Thus, the semantics of □ in GV in this extension differs to Granule which is 'pervasively' call-by-value, evaluating inside of modalities.

For more detail, Appendix E gives a rendering of replicate in Granule, but using its call-by-name language extension for soundness.

Non-linear communication Possibly the closest ideas to the ones described in our work that exist in the literature appear in the type system developed by Zalakain and Dardha, who use leftover typing to define a resource-aware variant of the π-calculus that can represent shared channels (which can be freely discarded or reused), graded channels (which must be used precisely n times for some n) and linear channels [33].

Applying their notion of 'graded channel' allows them to capture roughly the same behaviour as our `forkNonLinear` primitive, but their work does not consider more general forms of grading (though their mechanisation allows for user-defined resource algebras). We capture some of the same ideas inside our unified graded approach, showing how different amounts of sharing can be characterised precisely with the interaction of linear, indexed, and graded types; our setting is also more general, since we work in a pre-existing programming language rather than developing a calculus specifically focused on communication.

Another paper which describes some similar non-linear communication patterns to our work is the generalisation of session types via adjoint logic presented by Pruiksma and Pfenning [34]. Here, they offer the capacity for both multicast communication and replicated servers, corresponding to our `forkMulticast` and `forkReplicate` primitives respectively. However, their system does not include any notion of grading, and so while they offer non-linear communication this is without the precise quantification allowed by grading in our system.

Classical session types As well as the session-based interpretation of intuitionistic linear logic mentioned above [6], it is also possible to understand the correspondence between communication and logic by relying on *classical* linear logic, as was first demonstrated by Wadler [8]. The differences between these two systems go beyond the superficial; it has been shown that there exists a class of processes which are typable following the classical interpretation but not via the intuitionistic interpretation, and that these are the processes which break the so-called *locality principle* for shared channels [35].

Recent work by Kokke, Morris and Wadler extends this interpretation through an approach which draws on notions from bounded linear logic [9]. While their notion of a shared channel corresponds closely to the one that we develop in our work (which may be unsurprising, since graded modal types are also built upon the foundation of bounded linear logic), their focus is more on accounting for non-determinism and races in a logical setting while preserving deadlock freedom. Their paper contrasts the system they develop with other related work on *manifest sharing* [36]; said work uses modal operators to describe the interaction between linear and shared processes, and has greater expressivity when it comes to non-linear communication, but this comes at the cost of reintroducing deadlocks in some circumstances.

Graded sessions in linear Haskell We explored combining grading with linear session types in Granule, but this could be extended to other settings. The linear types extension to Haskell is based on a calculus involving graded function arrows [20], and adding additional multiplicities to this system is a possibility. The Priority Sesh library provides a convenient embedding of the GV linear session calculus in Haskell [37], and extending this to make use of more precise information about channel usage could be valuable. This would also allow for experimenting with graded channels in a setting where grading can be implicit, rather than one where (like Granule) all grades are explicitly encoded via modalities.

11. Discussion and conclusion

The key idea here is that graded types capture various standard non-linear forms of communication pattern atop the usual linear session types presentation. The demands of call-by-value required a different approach to past work on the interaction between graded types and session types, including the syntactic restriction discussed in Section 4 and vectors to capture multiplicity of channels. We have also proven type safety and various essential communication properties given the additional restriction on promoting channels and the other features introduced here.

Deadlock freedom Whilst the proofs of progress and preservation go some way to ensuring that our new primitives do not undermine the guarantees of the language, a further useful and important property to prove would be *deadlock freedom*. Whilst progress and preservation suggest that we don't become locally stuck, they do not rule out the possibility of some subset of processes becoming indefinitely blocked due to a mutual dependency. Proving deadlock freedom is future work though we sketch some details here based on other approaches in the literature.

There have been various proofs of deadlock freedom for session-typed calculi with asynchronous communication. These approaches generalise duality to *compatibility*, for example in the work of Fowler [38] and Bocchi et al. [39]. In essence, compatibility needs to take into account that asynchronous protocols may be 'out of sync' as one process sends to another, filling a channel's buffer without those messages being received yet. We can tie together the types of two channel endpoints P_1, P_2 with the contents of the associated channel values (buffers) $\vec{v}_1, \vec{v}_2$ via a relation:

$$\Delta \vdash c_1 : (P_1 - \vec{v}_1) \bowtie c_2 : (P_2 - \vec{v}_2)$$

meaning that P_1 is compatible with P_2 once the channel buffer values $\vec{v}_1$ and $\vec{v}_2$ have been 'stripped off' the head of the protocol. For example, a receiving protocol $\mathbf{Recv}\, A\, P$ for channel endpoint c is compatible with another protocol P' for the dual channel endpoint $\overline{c}$, if the channel has a value v of type A in its head, and then P and P' are compatible with that v removed:

$$\frac{\Delta \vdash c : (P - \vec{v}_1) \bowtie \overline{c} : (P' - \vec{v}_2) \qquad \Delta \vdash v : A}{\Delta \vdash c : ((\mathbf{Recv}\, A\, P) - v : \vec{v}_1) \bowtie \overline{c} : (P' - \vec{v}_2)} \bowtie_{\text{RECV}}$$

Following Fowler [38], we then need to adapt process configuration typing into a typing of the entire configuration (process and channels), with parallel composition enforcing that between two processes there is exactly one pair of compatible endpoints connecting them, i.e.:

$$\frac{\begin{array}{c}\Gamma_1, \Delta_1, c : \mathbf{Chan}\, P \vdash t : A \qquad \Gamma_2, \Delta_2, \overline{c} : \mathbf{Chan}\, P' \vdash \mathcal{P} : \bot ; \mathcal{C} \\ \Delta \vdash c : (P - \vec{v}_1) \bowtie \overline{c} : (P' - \vec{v}_2)\end{array}}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 + \Delta, c : \mathbf{Chan}\, P, \overline{c} : \mathbf{Chan}\, P' \vdash \mathcal{P} \mid t^{\bullet} : A ; \mathcal{C}, c = \vec{v}_1, \overline{c} = \vec{v}_2}$$

From this 'global' configuration typing, we can then prove that there is a limit on the number of 'connection' points between processes:

Definition 4 *(Connection).* For processes $\mathcal{P}_1$ and $\mathcal{P}_2$ then we define those channels endpoints that connect them via the function:

$$\mathsf{connected}(\mathcal{P}_1, \mathcal{P}_2) = \bigcup \{\{c, \overline{c}\} \mid c \in \mathsf{fn}(\mathcal{P}_1), \overline{c} \in \mathsf{fn}(\mathcal{P}_2)\}$$

Lemma 2 *(Limited connectivity). If* $\Gamma, \Delta \vdash \mathcal{P} : A; \mathcal{C}$ *and* $\mathcal{P} = \mathcal{P}' \mid \mathcal{P}_1 \mid \mathcal{P}_2$ *then* $\exists c.\mathsf{connections}(\mathcal{P}_1, \mathcal{P}_2) \subseteq \{c, \overline{c}\}$

We note however that, for a full proof for our system here, the above lemma needs extending to handle forwarding processes which have a number of channels connecting processes, but where only sending is allowed.

Deadlock can then be characterised in terms of mutually dependent blocked processes, which are ruled out by the typing, similar to the formulation of Lindley and Morris for GV [11]. That is, a term t blocked on a channel c is a term which is equivalent to some evaluation context in which a receive is applied to c:

$$\mathsf{blocked}(t, c) \triangleq \exists Ctxt.\ t \equiv Ctxt\{\mathsf{receive}\, c\} \vee t \equiv Ctxt\{\mathsf{close}\, c\}$$

Thus, the term t can only reduce further in composition with another a global configuration. A dependency can then be characterised as a term which is both blocked on a channel and with a free-variable dependency:

$$\begin{aligned}
\mathsf{depends}(t, c, x) &\triangleq \mathsf{blocked}(t, c) \ \wedge\ x \in \mathsf{fv}(Ctxt\{t\}) \\
\mathsf{depends}(\mathcal{P}, c, x) &\triangleq (\exists t, \mathcal{P}'.\ \mathcal{P} \equiv (\mathcal{P}' \mid t^{\mathcal{R}}) \ \wedge\ \mathsf{depends}(t, c, x)) \\
&\quad \vee (\exists \mathcal{P}_1, \mathcal{P}_2.\ \mathcal{P} \equiv (\mathcal{P}_1 \mid \mathcal{P}_2) \ \wedge\ \exists d.\, \mathsf{depends}(\mathcal{P}_1, c, d) \wedge \mathsf{depends}(\mathcal{P}_2, d, x))
\end{aligned}$$

The main definition of a deadlocked configuration with cyclic dependencies can then be characterised as:

$$\begin{aligned}
\mathsf{deadlocked}(\mathcal{P}) &\triangleq \exists \mathcal{P}', \mathcal{P}_1, \mathcal{P}_2.\ \mathcal{P} \equiv (\mathcal{P}' \mid \mathcal{P}_1 \mid \mathcal{P}_2) \\
&\quad \wedge\ \exists c, d.\mathsf{depends}(\mathcal{P}_1, c, d) \wedge \mathsf{depends}(\mathcal{P}_2, d, c)
\end{aligned}$$

From Lemma 2, we conjecture that this then enables the proof of deadlock freedom theorem:

Theorem conjecture 1. *If* $\Gamma, \Delta \vdash \mathcal{P} : A; \mathcal{C}$ *then* $\neg\mathsf{deadlocked}(\mathcal{P})$.

Further work remains to flesh out the details of Lemma 2 for forwarding, and to integrate the global configuration typing (processes and channels together, with compatibility) into the progress and preservation theorems here, rather than their current formulation with process and channel configuration typing separate.

Recursion and other combinators A notable omission from our core session types calculus is the ability to define *recursive protocols*. However, some of the power of recursive session types is provided here; reusable channels (Section 6) are equivalent to linearly recursive session types (e.g., $\mu x.P.x$), although the expected usage has to be precisely specified in the grading since for example allowing the grade $\mathtt{r}$ to be instantiated to $\mathtt{0..\infty}$ to capture an arbitrary amount of use would break our guarantee of deadlock freedom. Further work is to integrate standard ideas on recursive session types and to explore their interaction with grading.

Note that combinations of the ideas here should also be possible; for example, combining multicast sending with reuse to get channels which we can repeatedly use to broadcast upon.

Applications of graded sessions Using the linear channels already present in Granule it is possible to represent functions that are *sequentially realizable* [40]; a sequentially realizable function is one which has outwardly pure behaviour but relies on a notion of local side effects which are contained within the body of the function. Looking into which such behaviours may be more easily expressed by introducing non-linearity via graded channels would be an avenue for future work.

More recently, the GV session type calculus has been extended to allow for *multiparty* session types [41], allowing for concurrent communication between several participants at a time. It has also been shown that this multiparty system is strictly more general than the original binary GV system. It would be interesting to explore an implementation of this in Granule, particularly with a view to introducing grading into this setting to see if it can increase expressivity further.

Other resource allocators As discussed in Section 4, it is necessary to restrict promotion of channels due to Granule's default call-by-value semantics, since omitting this restriction allows for a linear channel to be used non-linearly. A very similar caveat applies if we consider mutable arrays. Recent work describes how to represent uniqueness of reference within Granule in addition to linearity [42]; however, this takes place in a call-by-name setting, because if a unique array is promoted under call-by-value then we end up with multiple references to the same array, so we can no longer guarantee uniqueness

for mutation. This is resolved in later work incorporating a call-by-value semantics by restricting promotion in much the same way as we have here for linear channels [43].

These are most likely not the only resource allocators for which such a restriction applies. Future work is to develop a better understanding of the general case here, and to further explore the best way to embed interfaces for allocating resources in a graded type system such as Granule's which can express multiple different evaluation strategies.

Uniqueness and graded sessions Uniqueness in a concurrent setting was considered in various works by de Vries et al., but in particular we draw attention to their paper on resource management via unique and affine channels [44]. They allow for a notion of a channel which we can guarantee unique access to after i communication steps where i is some natural number; this is a form of grading, though quite different from the ideas we have discussed. It would be interesting to explore how we can represent this in Granule which already has both uniqueness and graded session types, and whether we gain further expressivity from doing so. This idea closely relates to recent work by Rocha and Caires on introducing mutable state into the classical linear logic interpretation of session types [45], through typing rules inspired by differential linear logic.

CRediT authorship contribution statement

Danielle Marshall: Writing – original draft, Writing – review & editing. **Dominic Orchard:** Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Thank you to the anonymous reviewers for their helpful comments and feedback that have improved this paper considerably. Thanks to the participants of PLACES 2022 for their comments on the workshop version of this paper and to the anonymous reviews of both versions of this paper. Thanks also to Paulo Torrens for his comments regarding resource allocation. Thanks to the rest of the Granule team for their feedback and overall work on the research vehicle that is the Granule programming language. Thank you to Sam Lindley and Laura Bocchi for further discussion. This work was supported by an EPSRC Doctoral Training Award EP/T518141/1 (Marshall) and EPSRC grant EP/T013516/1 (*Verifying Resource-like Data Use in Programs via Types*). The second author also received support through Schmidt Sciences, LLC.

Appendix A. Collected rules

A.1. Typing

$$\frac{}{x : A \vdash x : A}\,\text{VAR} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\,\multimap_i \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\,\multimap_e$$

$$\frac{\Gamma_1 \vdash t_1 : A \quad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\,\otimes_i \qquad \frac{\Gamma_1 \vdash t_1 : A \otimes A' \quad \Gamma_2, x : A, y : A' \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ (x, y) = t_1\ \textbf{in}\ t_2 : B}\,\otimes_e$$

$$\frac{}{\emptyset \vdash \textbf{unit} : 1}\,\text{UNIT}_i \qquad \frac{\Gamma_1 \vdash t_1 : 1 \quad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let unit} = t_1\ \textbf{in}\ t_2 : B}\,\text{UNIT}_e$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\,\text{DER} \qquad \frac{\neg \mathsf{resourceAllocator}(t) \quad [\Gamma] \vdash t : A}{r * \Gamma \vdash [t] : \Box_r A}\,\text{PR} \qquad \frac{\Gamma \vdash t : B}{\Gamma, 0 * \Gamma' \vdash t : B}\,\text{WEAK}$$

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \quad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\,\text{APPROX}$$

Definition 5 *(Graded contexts).* $[\Gamma]$ classifies those contexts which contain only graded variables:

$$\frac{}{[\emptyset]} \qquad \frac{[\Gamma]}{[\Gamma, x : [A]_r]}$$

Definition 6 *(Resource allocating terms).* Predicate definition:

$$\frac{}{\mathsf{resourceAllocator}(\mathsf{forkLinear})} \qquad \frac{}{\mathsf{resourceAllocator}(\mathsf{forkNonLinear})}$$

$$\frac{}{\mathsf{resourceAllocator}(\mathsf{forkReplicate})} \qquad \frac{}{\mathsf{resourceAllocator}(\mathsf{forkMulticast})}$$

$$\frac{\mathsf{resourceAllocator}(t)}{\mathsf{resourceAllocator}([t])}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}((\lambda x.t_1)\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{let}\ [x]\ = t_1\ \mathbf{in}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\mathbf{let}\ [x]\ = t_1\ \mathbf{in}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{let}\ \mathbf{unit} = t_1\ \mathbf{in}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\mathbf{let}\ \mathbf{unit} = t_1\ \mathbf{in}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}((t_1, t_2))} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}((t_1, t_2))}$$

Primitives

$$\frac{}{\emptyset \vdash \mathsf{send} : A \otimes \mathbf{Chan}\,\mathbf{Send}\,A\,P \multimap \mathbf{Chan}\,P}\ \textsc{send}$$

$$\frac{}{\emptyset \vdash \mathsf{receive} : \mathbf{Chan}\,\mathbf{Recv}\,A\,P \multimap A \otimes \mathbf{Chan}\,P}\ \textsc{recv}$$

$$\frac{}{\emptyset \vdash \mathsf{forkLinear} : (\mathbf{Chan}\,P \multimap 1) \multimap \mathbf{Chan}\,\overline{P}}\ \textsc{fork}$$

$$\frac{}{\emptyset \vdash \mathsf{close} : \mathbf{Chan}\,\mathbf{End} \multimap 1}\ \textsc{close}$$

$$\frac{\mathsf{SingleAction}(P) \qquad \mathsf{ExactSemiring}(S) \qquad r : S}{\emptyset \vdash \mathsf{forkNonLinear} : (\Box_r \mathbf{Chan}\,P \multimap 1) \multimap \Box_r(\mathbf{Chan}\,\overline{P})}\ \textsc{forkNonLin}$$

$$\frac{\mathsf{ReceivePrefix}(P)}{\emptyset \vdash \mathsf{forkReplicate} : \Box_{0..n}(\mathbf{Chan}\,P \multimap 1) \multimap \mathbf{N}\,n \multimap \mathbf{Vec}\,n\,(\Box_{0..1}(\mathbf{Chan}\,\overline{P}))}\ \textsc{forkRep}$$

$$\frac{\mathsf{Sends}(P)}{\emptyset \vdash \mathsf{forkMulticast} : (\mathbf{Chan}\,(\mathsf{Graded}_n P) \multimap 1) \multimap \mathbf{N}\,n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})}\ \textsc{forkMulti}$$

Runtime typing

$$\frac{}{c : \mathbf{Chan}\,P \vdash c : \mathbf{Chan}\,P}\ \textsc{var} \qquad \frac{}{c :^* \mathbf{Chan}\,P \vdash c : \mathbf{Chan}\,P}\ \textsc{varChanS}$$

$$\frac{\Gamma \vdash t : B}{\Gamma, 0 * \Gamma', c :^* \mathbf{Chan}\,P \vdash t : B}\ \textsc{weakChanS}$$

In typing derivations in the appendix, we abstract over whether a channel is linear or shared by writing $c :^r \mathbf{Chan}\,P$.

A.2. Global configuration reduction rules

$$\frac{t \rightsquigarrow t'}{(\mathcal{P} \mid t^{\mathcal{R}};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid t'^{\mathcal{R}};\ \mathcal{C})}\text{REDUCE}$$

$$\frac{(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')}{(\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t;\ \mathcal{C}')}\text{PAR}$$

$$\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{send}\, c\, v\};\ \mathcal{C}, \overline{c} = \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{c\};\ \mathcal{C}, \overline{c} = \vec{v} : v)}\text{SEND}$$

$$\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{receive}\, c\};\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{(v, c)\};\ \mathcal{C}, c = \vec{v})}\text{RECV}$$

$$\frac{\#c}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkLinear}\,(\lambda x.t)\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^{\circ} \mid \mathit{Ctxt}\{\overline{c}\};\ \mathcal{C}, \overline{c} = \langle\rangle, c = \langle\rangle)}\text{FORKLINEAR}$$

$$\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{close}\, c\};\ \mathcal{C}, c^{*} = \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{\mathbf{unit}\};\ \mathcal{C}, c^{*} = \vec{v})}\text{SHAREDCLOSE}$$

$$\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{close}\, c\};\ \mathcal{C}, c^{1} = \langle\rangle) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{\mathbf{unit}\};\ \mathcal{C})}\text{CLOSE}$$

$$\frac{\#c}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkNonLinear}\,(\lambda x.t)\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{[\overline{c}]\} \mid t[[c]/x]^{\circ};\ \mathcal{C}, \overline{c}^{*} = \langle\rangle, c^{*} = \langle\rangle)}\text{FORKNONLINEAR}$$

$$\frac{\#c_1 \ldots \#c_n}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkReplicate}\,[\lambda x.t]\, n\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{(\overline{[c_1]}, \ldots, \overline{[c_n]})\} \mid \mathsf{Wch}\,(c_1 \mapsto \lambda x.t) \mid \ldots \mid \mathsf{Wch}\,(c_n \mapsto \lambda x.t);\ \mathcal{C}, c_1 = \langle\rangle, \ldots, c_n = \langle\rangle, \overline{c_1} = \langle\rangle, \ldots, \overline{c_n} = \langle\rangle)}\text{FORKREPLICATE}$$

$$\frac{\#c \quad \#c_1 \ldots \#c_n}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkMulticast}\,(\lambda x.t)\, n\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{(\overline{c_1}, \ldots, \overline{c_n})\} \mid t[c/x]^{\circ} \mid \mathsf{Fwd}\,(c \mapsto c_1, \ldots, c_n);\ \mathcal{C}, c = \langle\rangle, \overline{c} = \langle\rangle, c_1 = \langle\rangle, \ldots, c_n = \langle\rangle, \overline{c_1} = \langle\rangle, \ldots, \overline{c_n} = \langle\rangle)}\text{FORKMULTICAST}$$

A.3. Standard small-step reduction rules

$$\frac{}{(\lambda x.t)\, v \rightsquigarrow t[v/x]}\rightsquigarrow_{\beta} \qquad \frac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_2}\rightsquigarrow_{\text{APP}} \qquad \frac{t \rightsquigarrow t'}{v\, t \rightsquigarrow v\, t'}\rightsquigarrow_{\text{prim}}$$

$$\frac{}{\mathbf{let}\ (x, y) = (v_1, v_2)\ \mathbf{in}\ t_3 \rightsquigarrow t_3[v_1/x][v_2/y]}\rightsquigarrow_{\otimes\beta}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2 \rightsquigarrow \mathbf{let}\ (x, y) = t_1'\ \mathbf{in}\ t_2}\rightsquigarrow_{\text{LET}\otimes}$$

$$\frac{t_1 \rightsquigarrow t_1'}{(t_1, t_2) \rightsquigarrow (t_1', t_2)}\rightsquigarrow_{\otimes\text{reduceL}} \qquad \frac{t_2 \rightsquigarrow t_2'}{(v, t_2) \rightsquigarrow (v, t_2')}\rightsquigarrow_{\otimes\text{reduceR}}$$

$$\frac{}{\mathbf{let}\ \mathbf{unit} = \mathbf{unit}\ \mathbf{in}\ t \rightsquigarrow t}\rightsquigarrow_{\beta\text{unit}} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\ \mathbf{unit} = t_1\ \mathbf{in}\ t_2 \rightsquigarrow \mathbf{let}\ \mathbf{unit} = t_1'\ \mathbf{in}\ t_2}\rightsquigarrow_{\text{LETunit}}$$

$$\frac{}{\mathbf{let}\ [x]\ = [v]\ \mathbf{in}\ t \rightsquigarrow t[v/x]}\rightsquigarrow_{\Box\beta} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\ [x]\ = t_1\ \mathbf{in}\ t_2 \rightsquigarrow \mathbf{let}\ [x]\ = t_1'\ \mathbf{in}\ t_2}\rightsquigarrow_{\text{LET}\Box}$$

$$\frac{t \rightsquigarrow t'}{[t] \rightsquigarrow [t']}\rightsquigarrow_{\Box\text{reduce}}$$

Appendix B. Auxiliary results

Linear and graded substitution lemmas have been previously proved by Orchard et al. [16] for the core calculus used here:

Lemma 1 *(Linear substitution). [16] Given* $\Gamma_1, x : A \vdash t_1 : B$ *and* $\Gamma_2 \vdash t_2 : A$ *then* $\Gamma_1 + \Gamma_2 \vdash t_1[t_2/x] : B$.

Lemma 2 *(Graded substitution). [16] Given* $\Gamma_1, x : [A]_r \vdash t_1 : B$ *and* $[\Gamma_2] \vdash t_2 : A$ *and* $\neg\mathsf{resourceAllocator}(t_2)$ *then* $\Gamma_1 + r * \Gamma_2 \vdash t_1[t_2/x] : B$.

Proof. Proof as in Orchard et al. [16], but where the additional premise of $\neg\mathsf{resourceAllocator}(t_2)$ is deployed in the promotion case. □

Definition 7 *(Evaluation contexts).*

$$
\begin{array}{lcll}
Ctxt & ::= & & \text{Contexts} \\
 & | & - & \\
 & | & Ctxt\ t & \\
 & | & v\ Ctxt & \\
 & | & [Ctxt] & \\
 & | & \mathbf{let}\ [x] = Ctxt\ \mathbf{in}\ t & \\
 & | & \mathbf{let}\ \mathbf{unit} = Ctxt\ \mathbf{in}\ t & \\
 & | & \mathbf{let}\ (x, y) = Ctxt\ \mathbf{in}\ t & \\
 & | & (Ctxt, t) & \\
 & | & (v, Ctxt) &
\end{array}
$$

Lemma 3 *(Non-resource allocator context).* $\neg\mathsf{resourceAllocator}(Ctxt\{t\}) \implies \neg\mathsf{resourceAllocator}(Ctxt\{x\})$

Proof. Trivial by the definition of $\mathsf{resourceAllocator}(t)$ and since x is a variable and not a resource allocator. □

Lemma 4 *(Non-resource allocator composition).* $\neg\mathsf{resourceAllocator}(Ctxt\{x\}) \wedge \neg\mathsf{resourceAllocator}(t) \implies \neg\mathsf{resourceAllocator}(Ctxt\{t\})$

Proof. By induction on the definition of contexts and the definition of $\mathsf{resourceAllocator}(t)$. □

Lemma 5 *(Contexts preserve resource allocator property).*

$$\mathsf{resourceAllocator}(t) \implies \mathsf{resourceAllocator}(Ctxt\{t\})$$

Proof. By induction on the grammar of context terms: the structure of context terms matches the inductive term structure of resourceAllocator.

- $Ctxt = -$ by identity;
- $Ctxt = Ctxt'\ t'$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}(Ctxt'\{t\}\ t')$ by the definition.
- $Ctxt = v\ Ctxt'$. By induction on the second premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}(v\ Ctxt'\{t\})$ by the definition.
- $Ctxt = [Ctxt']$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}([Ctxt'\{t\}])$ by the definition.
- $Ctxt = \mathbf{let}\ [x] = Ctxt'\ \mathbf{in}\ t_2$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}(\mathbf{let}\ [x] = Ctxt'\{t\}\ \mathbf{in}\ t_2)$ by the definition.
- $Ctxt = \mathbf{let}\ \mathbf{unit} = Ctxt'\ \mathbf{in}\ t_2$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}(\mathbf{let}\ \mathbf{unit} = Ctxt'\{t\}\ \mathbf{in}\ t_2)$ by the definition.
- $Ctxt = \mathbf{let}\ (x, y) = Ctxt'\ \mathbf{in}\ t_2$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}(\mathbf{let}\ (x, y) = Ctxt'\{t\}\ \mathbf{in}\ t_2)$ by the definition.
- $Ctxt = (Ctxt', t')$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}((Ctxt'\{t\}, t'))$ by the definition.
- $Ctxt = (v, Ctxt')$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}((v, Ctxt'\{t\}))$ by the definition. □

Corollary 1 *(Non-resource-allocating contextual term).*

$$\neg\mathsf{resourceAllocator}(Ctxt\{t\}) \implies \neg\mathsf{resourceAllocator}(t)$$

Proof. Dual of Lemma 5 (modus tollens) □

Lemma 6 *(Context factorisation).*

$$\begin{aligned}
&\Gamma \vdash \mathit{Ctxt}\{t\} : B \\
\implies\ & (\exists \Gamma_1, \Gamma_2, A.\ \Gamma_1, x : A \vdash \mathit{Ctxt}\{x\} : B \\
&\qquad \wedge\ \Gamma_2 \vdash t : A \\
&\qquad \wedge\ \Gamma = \Gamma_1 + \Gamma_2) \\
\vee\ & (\exists \Gamma_1, \Gamma_2, A, r.\ \neg \mathsf{resourceAllocator}(t) \\
&\qquad \wedge\ \Gamma_1, x : [A]_r \vdash \mathit{Ctxt}\{x\} : B \\
&\qquad \wedge\ [\Gamma_2] \vdash t : A \\
&\qquad \wedge\ \Gamma = \Gamma_1 + r * \Gamma_2)
\end{aligned}$$

Proof. By induction on *Ctxt*:

- $\mathit{Ctxt} = -$, i.e., the premise is that $\Gamma \vdash t : B$ and thus the result is given by $\Gamma_1 = \cdot$ and $\Gamma_2 = \Gamma$ with derivation:

$$\textsc{var}\ x : B \vdash x : B \tag{ctxt}$$

  and the substitute derivation given by the premise. Thus, we prove this case by giving the first disjunct.
- $\mathit{Ctxt} = \mathit{Ctxt}'\ t'$ where $\mathit{Ctxt}\{t\} = \mathit{Ctxt}'\{t\}\ t'$ i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash \mathit{Ctxt}'\{t\} : A' \multimap B \qquad \Gamma_2' \vdash t' : A'}{\Gamma \vdash \mathit{Ctxt}'\{t\}\ t' : B} \multimap_e$$

  where $\Gamma = \Gamma_1' + \Gamma_2'$.
  By induction on the first premise, then there are two cases:
  – (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash \mathit{Ctxt}'\{x\} : A' \multimap B$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + \Gamma_2''$.
  We conclude with derivation:

$$\frac{\Gamma_1'', x : A \vdash \mathit{Ctxt}'\{x\} : A' \multimap B \qquad \Gamma_2' \vdash t' : A'}{(\Gamma_1'' + \Gamma_2'), x : A \vdash \mathit{Ctxt}'\{x\}\ t' : B} \multimap_e \tag{ctxt}$$

  i.e., where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ where $\Gamma = \Gamma_1 + \Gamma_2 = \Gamma_1'' + \Gamma_2' + \Gamma_2'' = \Gamma_1' + \Gamma_2'$ from the induction and by commutativity and associativity of $+$.
  The concluding substitute $\Gamma_2'' \vdash t : A$ is from the induction.
  – (second disjunct) We have Γ_1', Γ_2'' such that $\neg\mathsf{resourceAllocator}(t)$ and $\Gamma_1'', x : [A]_r \vdash \mathit{Ctxt}'\{x\} : A' \multimap B$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + r * \Gamma_2''$.
  We conclude with derivation:

$$\frac{\Gamma_1'', x : [A]_r \vdash \mathit{Ctxt}'\{x\} : A' \multimap B \qquad \Gamma_2' \vdash t' : A'}{(\Gamma_1'' + \Gamma_2'), x : A \vdash \mathit{Ctxt}'\{x\}\ t' : B} \multimap_e \tag{ctxt}$$

  i.e., where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and where $\Gamma_2 = [\Gamma_2'']$ where:

$$\begin{aligned}
\Gamma &= \Gamma_1' + \Gamma_2' && \{\text{premise}\} \\
&= \Gamma_1'' + r * \Gamma_2'' + \Gamma_2' && \{\text{induction } \Gamma_1'\} \\
&= \Gamma_1'' + \Gamma_2' + r * \Gamma_2'' && \{+ \text{ comm. and assoc.}\} \\
&= \Gamma_1 + r * \Gamma_2 && \{\text{concluding contexts}\}
\end{aligned}$$

  The concluding substitute $[\Gamma_2''] \vdash t : A$ is from the induction and that $\neg\mathsf{resourceAllocator}(t)$.
- $\mathit{Ctxt} = v\,\mathit{Ctxt}'$ where $\mathit{Ctxt}\{t\} = v\,\mathit{Ctxt}'\{t\}$ i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash v : A' \multimap B \qquad \Gamma_2' \vdash \mathit{Ctxt}'\{t\} : A'}{\Gamma \vdash v\,\mathit{Ctxt}'\{t\} : B} \multimap_e$$

  where $\Gamma = \Gamma_1' + \Gamma_2'$.
  By induction on the second premise, then there are two cases:
  – (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash \mathit{Ctxt}'\{x\} : A'$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_2' = \Gamma_1'' + \Gamma_2''$. We can then conclude with:

$$\frac{\Gamma_1' \vdash v : A' \multimap B \qquad \Gamma_1'', x : A \vdash \mathit{Ctxt}'\{x\} : A'}{\Gamma_1' + \Gamma_1'', x : A \vdash v\,\mathit{Ctxt}'\{x\} : B} \multimap_e \tag{ctxt}$$

  and substitute $\Gamma_2'' \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and $\Gamma_2 = \Gamma_2''$ where $\Gamma = \Gamma_1 + \Gamma_2 = \Gamma_1' + \Gamma_1'' + \Gamma_2''$ from the induction and by commutativity and associativity of $+$.

– (second disjunct) We have Γ_1'', Γ_2'' such that $\neg\mathsf{resourceAllocator}(t)$ and $\Gamma_1'', x : [A]_r \vdash Ctxt'\{x\} : A'$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_2' = \Gamma_1'' + r * \Gamma_2''$.
We can then conclude with:

$$\frac{\Gamma_1' \vdash v : A' \multimap B \qquad \Gamma_1'', x : [A]_r \vdash Ctxt'\{x\} : A'}{\Gamma_1' + \Gamma_1'', x : [A]_r \vdash v\, Ctxt'\{x\} : B} \multimap_e \tag{ctxt}$$

and $\neg\mathsf{resourceAllocator}(t)$ (from the induction) and substitute $[\Gamma_2''] \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and $\Gamma_2 = \Gamma_2''$ where:

$$\begin{aligned} \Gamma &= \Gamma_1' + \Gamma_2' && \{\text{premise }\} \\ &= \Gamma_1' + \Gamma_1'' + r * \Gamma_2'' && \{\text{induction } \Gamma_1'\ \} \\ &= \Gamma_1 + r * \Gamma_2 && \{\text{concluding contexts}\} \end{aligned}$$

- $Ctxt = [Ctxt']$ where $Ctxt\{t\} = [Ctxt'\{t\}]$ i.e., with premise derivation:

$$\frac{\neg\mathsf{resourceAllocator}(Ctxt'\{t\}) \qquad [\Gamma_1'] \vdash Ctxt'\{t\} : A'}{\Gamma \vdash [Ctxt'\{t\}] : \Box_r A'}\ \mathrm{PR},$$

where $\Gamma = r * \Gamma_1'$.
By induction on the second premise, there are two cases:
– (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash Ctxt'\{x\} : A'$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + \Gamma_2''$, and we know that $[\Gamma_1'']$ since the premise derivation gives us that $[\Gamma_1']$.
We can then conclude with the derivation (giving the second disjunct):

$$\frac{\dfrac{\textit{Lemma 3}}{\neg\mathsf{resourceAllocator}(Ctxt'\{x\})} \qquad \dfrac{\dfrac{\text{ind.hyp.}}{[\Gamma_1''], x : A \vdash Ctxt'\{x\} : A'}}{[\Gamma_1''], x : [A]_1 \vdash Ctxt'\{x\} : A'}\ \mathrm{DER}}{r * \Gamma_1'', x : [A]_r \vdash [Ctxt'\{x\}] : \Box_r A'}\ \mathrm{PR} \tag{ctxt}$$

with substitute $\Gamma_2'' \vdash t : A$ (from the induction) and with $\Gamma_1 = r * \Gamma_1''$ and $\Gamma_2 = \Gamma_2''$ where:

$$\begin{aligned} \Gamma &= r * \Gamma_1' && \{\text{premise }\} \\ &= r * (\Gamma_1'' + \Gamma_2'') && \{\text{induction } \Gamma_1'\ \} \\ &= r * \Gamma_1'' + r * \Gamma_2'' && \{* \text{ dist. }\} \\ &= \Gamma_1 + r * \Gamma_2 && \{\text{concluding contexts}\} \end{aligned}$$

From the premise derivation $\neg\mathsf{resourceAllocator}(Ctxt'\{t\})$, by Corollary 1, we then have that $\neg\mathsf{resourceAllocator}(t)$ as needed here.
– (second disjunct) We have $\neg\mathsf{resourceAllocator}(t)$ and Γ_1'', Γ_2'' such that $\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : A'$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + s * \Gamma_2''$, and we know that $[\Gamma_1'']$ since the premise derivation gives us that $[\Gamma_1']$.
We can then conclude with the derivation (giving the second disjunct):

$$\frac{\dfrac{\textit{Lemma 3}}{\neg\mathsf{resourceAllocator}(Ctxt'\{x\})} \qquad \dfrac{\text{ind.hyp.}}{[\Gamma_1''], x : [A]_s \vdash Ctxt'\{x\} : A'}}{r * \Gamma_1'', x : [A]_{r*s} \vdash [Ctxt'\{x\}] : \Box_r A'}\ \mathrm{PR} \tag{ctxt}$$

with substitute $[\Gamma_2''] \vdash t : A$ (from the induction) and with $\Gamma_1 = r * \Gamma_1''$ and $\Gamma_2 = \Gamma_2''$ where:

$$\begin{aligned} \Gamma &= r * \Gamma_1' && \{\text{premise }\} \\ &= r * (\Gamma_1'' + s * \Gamma_2'') && \{\text{induction } \Gamma_1'\ \} \\ &= r * \Gamma_1'' + r * s * \Gamma_2'' && \{* \text{ dist. }\} \\ &= \Gamma_1 + r * s * \Gamma_2 && \{\text{concluding contexts}\} \end{aligned}$$

and $\neg\mathsf{resourceAllocator}(t)$ from the induction, as required.
- $Ctxt = \mathbf{let}\,[y] = Ctxt'\,\mathbf{in}\,t_2$ where $Ctxt\{t\} = \mathbf{let}\ [y]\ = Ctxt'\{t\}\ \mathbf{in}\ t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash Ctxt'\{t\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{\Gamma_1' + \Gamma_2' \vdash \mathbf{let}\ [y]\ = Ctxt'\{t\}\ \mathbf{in}\ t_2 : B}\ \mathrm{ELIM}$$

where $\Gamma = \Gamma_1' + \Gamma_2'$.
By induction on the first premise, then there are two case:

– (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash Ctxt'\{x\} : A'$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + \Gamma_2''$.
We can then conclude with:

$$\frac{\Gamma_1'', x : A \vdash Ctxt'\{x\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{(\Gamma_1'' + \Gamma_2'), x : A \vdash \textbf{let}\ [y]\ = Ctxt'\{x\}\ \textbf{in}\ t_2 : B}\ \text{ELIM}$$

and substitute $\Gamma_2'' \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{array}{ll} \Gamma = & \Gamma_1' + \Gamma_2' \qquad \{\text{premise }\} \\ = & \Gamma_1'' + \Gamma_2'' + \Gamma_2' \ \{\text{induction } \Gamma_1'\ \} \\ = & \Gamma_1'' + \Gamma_2' + \Gamma_2'' \ \{+ \text{ comm. and assoc.}\} \\ = & \Gamma_1 + \Gamma_2 \qquad \{\text{concluding contexts}\} \end{array}$$

– (second disjunct) We have $\neg\mathsf{resourceAllocator}(t)$ and Γ_1'', Γ_2'' such that $\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : A'$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + s * \Gamma_2''$.
We can then conclude with:

$$\frac{\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{(\Gamma_1'' + \Gamma_2'), x : [A]_s \vdash \textbf{let}\ [y]\ = Ctxt'\{x\}\ \textbf{in}\ t_2 : B}\ \text{ELIM}$$

and substitute $[\Gamma_2''] \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{array}{ll} \Gamma = & \Gamma_1' + \Gamma_2' \qquad \{\text{premise }\} \\ = & \Gamma_1'' + s * \Gamma_2'' + \Gamma_2' \ \{\text{induction } \Gamma_1'\ \} \\ = & \Gamma_1'' + \Gamma_2' + s * \Gamma_2'' \ \{+ \text{ comm. and assoc.}\} \\ = & \Gamma_1 + s * \Gamma_2 \qquad \{\text{concluding contexts}\} \end{array}$$

and $\neg\mathsf{resourceAllocator}(t)$ as required, from the induction.

- $Ctxt = \textbf{let unit} = Ctxt'\ \textbf{in}\ t_2$ where $Ctxt\{t\} = \textbf{let unit} = Ctxt'\{t\}\ \textbf{in}\ t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash Ctxt'\{t\} : 1 \qquad \Gamma_2' \vdash t_2 : B}{\Gamma_1' + \Gamma_2' \vdash \textbf{let unit} = Ctxt'\{t\}\ \textbf{in}\ t_2 : B}\ \text{ELIM}$$

where $\Gamma = \Gamma_1' + \Gamma_2'$.
By induction on the first premise, then there are two cases:
– (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash Ctxt'\{x\} : 1$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + \Gamma_2''$.
We can then conclude with:

$$\frac{\Gamma_1'', x : A \vdash Ctxt'\{x\} : 1 \qquad \Gamma_2' \vdash t_2 : B}{(\Gamma_1'' + \Gamma_2'), x : A \vdash \textbf{let}\ [y]\ = Ctxt'\{x\}\ \textbf{in}\ t_2 : B}\ \text{ELIM}$$

and substitute $\Gamma_2'' \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{array}{ll} \Gamma = & \Gamma_1' + \Gamma_2' \qquad \{\text{premise }\} \\ = & \Gamma_1'' + \Gamma_2'' + \Gamma_2' \ \{\text{induction } \Gamma_1'\ \} \\ = & \Gamma_1'' + \Gamma_2' + \Gamma_2'' \ \{+ \text{ comm. and assoc.}\} \\ = & \Gamma_1 + \Gamma_2 \qquad \{\text{concluding contexts}\} \end{array}$$

– (second disjunct) We have $\neg\mathsf{resourceAllocator}(t)$ and Γ_1'', Γ_2'' such that $\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : 1$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + s * \Gamma_2''$.
We can then conclude with:

$$\frac{\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : 1 \qquad \Gamma_2' \vdash t_2 : B}{(\Gamma_1'' + \Gamma_2'), x : [A]_s \vdash \textbf{let}\ [y]\ = Ctxt'\{x\}\ \textbf{in}\ t_2 : B}\ \text{ELIM}$$

and substitute $[\Gamma_2''] \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{array}{ll} \Gamma = & \Gamma_1' + \Gamma_2' \qquad \{\text{premise }\} \\ = & \Gamma_1'' + s * \Gamma_2'' + \Gamma_2' \ \{\text{induction } \Gamma_1'\ \} \\ = & \Gamma_1'' + \Gamma_2' + s * \Gamma_2'' \ \{+ \text{ comm. and assoc.}\} \\ = & \Gamma_1 + s * \Gamma_2 \qquad \{\text{concluding contexts}\} \end{array}$$

and $\neg\mathsf{resourceAllocator}(t)$ as required, from the induction.

- $Ctxt = \textbf{let}\,(x, y) = Ctxt\ \textbf{in}\ t_2$ the same as above, modulo the additional free variables for x and y in the scope of t_2;

- $Ctxt = (Ctxt', t')$ where $Ctxt\{t\} = (Ctxt'\{t\}, t')$. i.e., with premise derivation:

$$\frac{\Gamma'_1 \vdash Ctxt'\{t\} : A' \qquad \Gamma'_2 \vdash t' : B'}{\Gamma'_1 + \Gamma'_2 \vdash (Ctxt'\{t\}, t') : A' \otimes B'} \otimes_i$$

  where $\Gamma = \Gamma'_1 + \Gamma'_2$.
  By induction on the first premise, then there are two case:
  - (first disjunct) We have Γ''_1, Γ''_2 such that $\Gamma''_1, x : A \vdash Ctxt'\{x\} : A'$ and $\Gamma''_2 \vdash t : A$ such that $\Gamma'_1 = \Gamma''_1 + \Gamma''_2$.
    We can then conclude with:

$$\frac{\Gamma''_1, x : A \vdash Ctxt'\{x\} : A' \qquad \Gamma'_2 \vdash t' : B'}{\Gamma''_1 + \Gamma'_2, x : A \vdash (Ctxt'\{t\}, t') : A' \otimes B'} \otimes_i$$

    and substitute $\Gamma''_2 \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma''_1 + \Gamma'_2$ and $\Gamma_2 = \Gamma''_2$ therefore:

$$\begin{array}{lll} \Gamma = & \Gamma'_1 + \Gamma'_2 & \{\text{premise }\} \\ = & \Gamma''_1 + \Gamma''_2 + \Gamma'_2 & \{\text{induction } \Gamma'_1\ \} \\ = & \Gamma''_1 + \Gamma'_2 + \Gamma''_2 & \{+ \text{ comm. and assoc.}\} \\ = & \Gamma_1 + \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

  - (second disjunct) We have $\neg\mathsf{resourceAllocator}(t)$ and Γ''_1, Γ''_2 such that $\Gamma''_1, x : [A]_s \vdash Ctxt'\{x\} : A'$ and $[\Gamma''_2] \vdash t : A$ such that $\Gamma'_1 = \Gamma''_1 + s * \Gamma''_2$.
    We can then conclude with:

$$\frac{\Gamma''_1, x : [A]_s \vdash Ctxt'\{x\} : A' \qquad \Gamma'_2 \vdash t' : B'}{\Gamma''_1 + \Gamma'_2, x : [A]_s \vdash (Ctxt'\{t\}, t') : A' \otimes B'} \otimes_i$$

    and substitute $[\Gamma''_2] \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma''_1 + \Gamma'_2$ and $\Gamma_2 = \Gamma''_2$ therefore:

$$\begin{array}{lll} \Gamma = & \Gamma'_1 + \Gamma'_2 & \{\text{premise }\} \\ = & \Gamma''_1 + s * \Gamma''_2 + \Gamma'_2 & \{\text{induction } \Gamma'_1\ \} \\ = & \Gamma''_1 + \Gamma'_2 + s * \Gamma''_2 & \{+ \text{ comm. and assoc.}\} \\ = & \Gamma_1 + s * \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

    and $\neg\mathsf{resourceAllocator}(t)$ as required, from the induction.
- $Ctxt = (v, Ctxt')$ where $Ctxt\{t\} = (v, Ctxt'\{t\})$. i.e., with premise derivation:

$$\frac{\Gamma'_1 \vdash v : A' \qquad \Gamma'_2 \vdash Ctxt'\{t\} : B'}{\Gamma'_1 + \Gamma'_2 \vdash (v, Ctxt'\{t\}) : A' \otimes B'} \otimes_i$$

  where $\Gamma = \Gamma'_1 + \Gamma'_2$.
  By induction on the second premise, then there are two case:
  - (first disjunct) We have Γ''_1, Γ''_2 such that $\Gamma''_1, x : A \vdash Ctxt'\{x\} : A'$ and $\Gamma''_2 \vdash t : A$ such that $\Gamma'_2 = \Gamma''_1 + \Gamma''_2$.
    We can then conclude with:

$$\frac{\Gamma'_1 \vdash v : A' \qquad \Gamma''_1, x : A \vdash Ctxt'\{x\} : B'}{(\Gamma'_1 + \Gamma''_1), x : A \vdash (v, Ctxt'\{x\}) : A' \otimes B'} \otimes_i$$

    and substitute $\Gamma''_2 \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma'_1 + \Gamma''_1$ and $\Gamma_2 = \Gamma''_2$ therefore:

$$\begin{array}{lll} \Gamma = & \Gamma'_1 + \Gamma'_2 & \{\text{premise }\} \\ = & \Gamma'_1 + \Gamma''_1 + \Gamma''_2 & \{\text{induction } \Gamma'_1\ \} \\ = & \Gamma_1 + \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

  - (second disjunct) We have $\neg\mathsf{resourceAllocator}(t)$ and Γ''_1, Γ''_2 such that $\Gamma''_1, x : [A]_s \vdash Ctxt'\{x\} : A'$ and $[\Gamma''_2] \vdash t : A$ such that $\Gamma'_2 = \Gamma''_1 + s * \Gamma''_2$.
    We can then conclude with:

$$\frac{\Gamma'_1 \vdash v : A' \qquad \Gamma''_1, x : [A]_s \vdash Ctxt'\{x\} : B'}{(\Gamma'_1 + \Gamma''_1), x : [A]_s \vdash (v, Ctxt'\{x\}) : A' \otimes B'} \otimes_i$$

    and substitute $[\Gamma''_2] \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma'_1 + \Gamma''_1$ and $\Gamma_2 = \Gamma''_2$ therefore:

$$\begin{array}{lll} \Gamma = & \Gamma'_1 + \Gamma'_2 & \{\text{premise }\} \\ = & \Gamma'_1 + \Gamma''_1 + s * \Gamma''_2 & \{\text{induction } \Gamma'_1\ \} \\ = & \Gamma_1 + s * \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

    and $\neg\mathsf{resourceAllocator}(t)$ as required, from the induction. □

Lemma 7 *(Context application (linear)).*

$$\begin{aligned} &\Gamma_1, x : A \vdash \mathit{Ctxt}\{x\} : B \\ \wedge\ &\Gamma_2 \vdash t : A \\ \Longrightarrow\ &\Gamma_1 + \Gamma_2 \vdash \mathit{Ctxt}\{t\} : B \end{aligned}$$

Proof. By induction on the structure of evaluation contexts

- $\mathit{Ctxt} = -$, i.e., the premise is:

$$\frac{}{x : A \vdash x : A}\ \text{VAR} \tag{ctxt}$$

  (i.e., $\Gamma_1 = \cdot$ and $A = B$).
  Then the conclusion follows by $\mathit{Ctxt}\{t\} = t$ therefore the second premise provides the conclusion.
- $\mathit{Ctxt} = \mathit{Ctxt}'\ t'$ where $\mathit{Ctxt}\{x\} = \mathit{Ctxt}'\{x\}\ t'$ i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash \mathit{Ctxt}'\{x\} : A' \multimap B \qquad \Gamma_2' \vdash t' : A'}{(\Gamma_1' + \Gamma_2'), x : A \vdash \mathit{Ctxt}'\{x\}\ t' : B}\ \multimap_e$$

  By induction on the first premise, yielding: $\Gamma_1' + \Gamma_2 \vdash \mathit{Ctxt}'\{t\} : A' \multimap B$ therefore the conclusion can be constructed as:

$$\frac{\Gamma_1' + \Gamma_2 \vdash \mathit{Ctxt}'\{t\} : A' \multimap B \qquad \Gamma_2' \vdash t' : A'}{(\Gamma_1' + \Gamma_2) + \Gamma_2' \vdash \mathit{Ctxt}'\{t\}\ t' : B}\ \multimap_e$$

  which by commutativity and associativity of $+$ matches the conclusion.
- $\mathit{Ctxt} = v\ \mathit{Ctxt}'$ where $\mathit{Ctxt}\{x\} = v\ \mathit{Ctxt}'\{x\}$ i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash v : A' \multimap B \qquad \Gamma_2', x : A \vdash \mathit{Ctxt}'\{x\} : A'}{(\Gamma_1' + \Gamma_2'), x : A \vdash v\ \mathit{Ctxt}'\{x\} : B}\ \multimap_e$$

  By induction on the second premise, yielding: $\Gamma_2' + \Gamma_2 \vdash \mathit{Ctxt}'\{t\} : A'$ therefore the conclusion can be constructed as:

$$\frac{\Gamma_1' \vdash v : A' \multimap B \qquad \Gamma_2' + \Gamma_2 \vdash \mathit{Ctxt}'\{t\} : A'}{(\Gamma_1' + \Gamma_2') + \Gamma_2 \vdash v\ \mathit{Ctxt}'\{t\} : B}\ \multimap_e$$

  satisfying the goal of the lemma.
- $\mathit{Ctxt} = [\mathit{Ctxt}']$ where $\mathit{Ctxt}\{x\} = [\mathit{Ctxt}'\{x\}]$ i.e., with premise derivation:

$$\frac{[\Gamma_1'] \vdash \mathit{Ctxt}'\{x\} : A'}{r * \Gamma_1' \vdash [\mathit{Ctxt}'\{x\}] : \Box_r A'}\ \text{PR}$$

  where Γ_1' must contain x. However, this leads to a contradiction since there cannot possibly be a linear assumption x in the conclusion due to the form of the typing. Therefore, the premise is false and this case is trivial.
- $\mathit{Ctxt} = \mathbf{let}\,[y] = \mathit{Ctxt}'\,\mathbf{in}\,t_2$ where $\mathit{Ctxt}\{x\} = \mathbf{let}\ [y]\ = \mathit{Ctxt}'\{x\}\ \mathbf{in}\ t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash \mathit{Ctxt}'\{x\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{(\Gamma_1' + \Gamma_2'), x : A \vdash \mathbf{let}\ [y]\ = \mathit{Ctxt}'\{x\}\ \mathbf{in}\ t_2 : B}\ \text{ELIM}$$

  By induction on the first premise, yielding: $\Gamma_1' + \Gamma_2 \vdash \mathit{Ctxt}'\{t\} : \Box_r A'$. Therefore the conclusion can be constructed as:

$$\frac{\Gamma_1' + \Gamma_2 \vdash \mathit{Ctxt}'\{t\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{(\Gamma_1' + \Gamma_2) + \Gamma_2' \vdash \mathbf{let}\ [y]\ = \mathit{Ctxt}'\{t\}\ \mathbf{in}\ t_2 : B}\ \text{ELIM}$$

  which by commutativity and associativity of $+$ matches the goal.
- $\mathit{Ctxt} = \mathbf{let}\,\mathbf{unit}\ = \mathit{Ctxt}'\,\mathbf{in}\,t_2$ where $\mathit{Ctxt}\{x\} = \mathbf{let}\ \mathbf{unit} = \mathit{Ctxt}'\{x\}\ \mathbf{in}\ t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash \mathit{Ctxt}'\{x\} : 1 \qquad \Gamma_2' \vdash t_2 : B}{(\Gamma_1' + \Gamma_2'), x : A \vdash \mathbf{let}\ [y]\ = \mathit{Ctxt}'\{x\}\ \mathbf{in}\ t_2 : B}\ \text{ELIM}$$

  By induction on the first premise, yielding: $\Gamma_1' + \Gamma_2 \vdash \mathit{Ctxt}'\{t\} : \Box_r A'$. Therefore the conclusion can be constructed as:

$$\frac{\Gamma_1' + \Gamma_2 \vdash \mathit{Ctxt}'\{t\} : 1 \qquad \Gamma_2' \vdash t_2 : B}{(\Gamma_1' + \Gamma_2) + \Gamma_2' \vdash \mathbf{let}\ [y]\ = \mathit{Ctxt}'\{t\}\ \mathbf{in}\ t_2 : B}\ \text{ELIM}$$

  which by commutativity and associativity of $+$ matches the goal.

- $Ctxt = \mathbf{let}\,(x, y) = Ctxt\,\mathbf{in}\,t_2$ the same as above, modulo the additional free variables for x and y in the scope of t_2;
- $Ctxt = (Ctxt', t')$ where $Ctxt\{x\} = (Ctxt'\{x\}, t')$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : A' \qquad \Gamma_2' \vdash t' : B'}{(\Gamma_1' + \Gamma_2'), x : A \vdash (Ctxt'\{x\}, t') : A' \otimes B'} \otimes_i$$

  By induction on the first premise then $\Gamma_1' + \Gamma_2 \vdash Ctxt'\{t\} : A'$. Therefore the goal can be constructed as:

$$\frac{\Gamma_1' + \Gamma_2 \vdash Ctxt'\{t\} : A' \qquad \Gamma_2' \vdash t' : B'}{(\Gamma_1' + \Gamma_2) + \Gamma_2' \vdash (Ctxt'\{x\}, t') : A' \otimes B'} \otimes_i$$

  satisfying the goal by commutativity and associativity of $+$.
- $Ctxt = (v, Ctxt')$ where $Ctxt\{x\} = (v, Ctxt'\{x\})$. i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash v : A' \qquad \Gamma_2', x : A \vdash Ctxt'\{x\} : B'}{(\Gamma_1' + \Gamma_2'), x : A \vdash (v, Ctxt'\{x\}) : A' \otimes B'} \otimes_i$$

  By induction on the second premise we have $\Gamma_2' + \Gamma_2 \vdash Ctxt'\{t\} : B'$. Therefore the goal can be constructed as:

$$\frac{\Gamma_1' \vdash v : A' \qquad \Gamma_2' + \Gamma_2 \vdash Ctxt'\{t\} : B'}{\Gamma_1' + (\Gamma_2' + \Gamma_2) \vdash (v, Ctxt'\{x\}) : A' \otimes B'} \otimes_i$$

  satisfying the goal by associativity of $+$. □

Lemma 8 *(Context application (graded)).*

$$\begin{aligned} &\Gamma_1, x : [A]_r \vdash Ctxt\{x\} : B \\ &\wedge\, [\Gamma_2] \vdash t : A \\ &\wedge\, \neg \mathsf{resourceAllocator}(t) \\ \Longrightarrow\ &\Gamma_1 + r * \Gamma_2 \vdash Ctxt\{t\} : B \end{aligned}$$

Proof. By induction on the structure of evaluation contexts and their typing.

Note that for all possible contexts, it is possible that typing is by (non-syntax directed) (approx) rule and thus we have:

$$\frac{\Gamma_1, x : [A]_s \vdash t' : B \qquad s \sqsubseteq r}{\Gamma_1, x : [A]_r \vdash t' : B}\ \text{APPROX}$$

Then by induction on the premise we have $\Gamma_1 + s * \Gamma_2 \vdash t' : B$.

Let $\Gamma_2 = x_i : [A_i]_{s_i}$ for $1 \le i \le |\Gamma_2|$ since by the premise every assumption is graded in the context. Then we can then construct the conclusion as follows:

$$\frac{\Gamma_1 + s * \Gamma_2 \vdash t' : B \qquad \dfrac{\dfrac{}{s_i \sqsubseteq s_i}\ \text{REFL} \qquad s \sqsubseteq r}{s * s_i \sqsubseteq r * s_i}\ \text{MONOTON.+UNIT}}{\Gamma_1 + r * \Gamma_2 \vdash t' : B}\ \text{APPROX}^*$$

The rest is by induction now we have considered this possibility at each context term:

- $Ctxt = -$.
  There are therefore two possibilities for the derivation of the premise:
  – (var/der combination)

$$\frac{\dfrac{}{x : A \vdash x : A}\ \text{VAR}}{x : [A]_1 \vdash x : A}\ \text{DER}$$

  (i.e., $\Gamma_1 = \cdot$ and $A = B$ and $r = 1$).
  Since $1 * \Gamma_2 = \Gamma_2$ and since $Ctxt\{t_2\} = t_2$ then we can conclude with the second premise $\Gamma_2 \vdash t_2 : B$

– (var/der/approx combination)

$$\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}\,\text{VAR}}{x : [A]_1 \vdash x : A}\,\text{DER} \quad 1 \sqsubseteq r}{x : [A]_r \vdash x : A}\,\text{APPROX}$$

(i.e., $\Gamma_1 = \cdot$ and $A = B$).
Let $\Gamma_2 = x_i : [A]_{s_i}$ (since by the premise every assumption is graded in the context). Then we can then construct the conclusion as follows:

$$\dfrac{\dfrac{}{\Gamma_2 \vdash t : A}\,(\text{SECOND PREMISE}) \qquad \dfrac{\dfrac{}{s_i \sqsubseteq s_i}\,\text{REFL} \quad 1 \sqsubseteq r}{s_i \sqsubseteq r * s_i}\,\text{MONOTON.+UNIT}}{r * \Gamma_2 \vdash t : A}\,\text{APPROX}^*$$

- $Ctxt = Ctxt'\ t'$ where $Ctxt\{x\} = Ctxt'\{x\}\ t'$. i.e., with premise derivation:

$$\dfrac{\Gamma_1', x : [A]_{s_1} \vdash Ctxt'\{x\} : A' \multimap B \qquad \Gamma_2', x : [A]_{s_2} \vdash t' : A'}{(\Gamma_1' + \Gamma_2'), x : [A]_{s_1 + s_2} \vdash Ctxt'\{x\}\ t' : B}\ \multimap_e$$

i.e., where $r = s_1 + s_2$. This derivation applies without loss of generality here for application since we can set $s2 = 0$ in the case that $x \notin \mathsf{FV}(t')$.
By induction on both premise we attain:

$$\Gamma_1' + s_1 * \Gamma_2 \vdash Ctxt'\{t\} : A' \multimap B \tag{ih1}$$

$$\Gamma_2' + s_2 * \Gamma_2 \vdash t' : A' \tag{ih2}$$

From which we construct the concluding judgment:

$$\dfrac{\Gamma_1' + s_1 * \Gamma_2 \vdash Ctxt'\{t\} : A' \multimap B \qquad \Gamma_2' + s_2 * \Gamma_2 \vdash t' : A'}{\Gamma_1' + s_1 * \Gamma_2 + \Gamma_2' + s_2 * \Gamma_2 \vdash Ctxt'\{t\}\ t' : B}\ \multimap_e$$

which since $r = s_1 + s_2$, and by distributivity of $*$ over $+$, and commutativity and associativity of $+$ yields: $\Gamma_1' + \Gamma_2' + r * \Gamma_2 \vdash Ctxt'\{t\}\ t' : B$.

- $Ctxt = v\ Ctxt'$ where $Ctxt\{x\} = v\ Ctxt'\{x\}$ i.e., with premise derivation:

$$\dfrac{\Gamma_1', x : [A]_{s_1} \vdash v : A' \multimap B \qquad \Gamma_2', x : [A]_{s_2} \vdash Ctxt'\{x\} : A'}{(\Gamma_1' + \Gamma_2'), x : [A]_{s_1 + s_2} \vdash v\ Ctxt'\{x\} : B}\ \multimap_e$$

This case proceeds as above by induction on both premises.

- $Ctxt = [Ctxt']$ where $Ctxt\{x\} = [Ctxt'\{x\}]$ i.e., with premise derivation:

$$\dfrac{\neg\mathsf{resourceAllocator}(Ctxt'\{x\}) \qquad [\Gamma_1'], x : [A]_{s_2} \vdash Ctxt'\{x\} : A'}{s_1 * (\Gamma_1', x : [A]_{s_2}) \vdash [Ctxt'\{x\}] : \Box_r A'}\ \text{PR'}$$

i.e., where $r = s_1 * s_2$ and $\Gamma_1 = s_1 * \Gamma_1'$
By induction on the second premise we attain:

$$\Gamma_1' + s_2 * \Gamma_2 \vdash Ctxt'\{t\} : A' \tag{ih}$$

From which we construct the concluding judgment:

$$\dfrac{\dfrac{\textit{Lemma 4}}{\neg\mathsf{resourceAllocator}(Ctxt'\{t\})} \qquad \Gamma_1' + s_2 * \Gamma_2 \vdash Ctxt'\{t\} : A'}{s_1 * (\Gamma_1' + s_2 * \Gamma_2) \vdash [Ctxt'\{t\}] : \Box_r A'}\ \text{PR'}$$

which by distributivity of $*$ over $+$ and associativity of $*$ yields: $s_1 * \Gamma_1' + (s_1 * s_2) * \Gamma_2 \vdash [Ctxt'\{t\}] : \Box_r A'$. and by $r = s_1 * s_2$ and $\Gamma_1 = s_1 * \Gamma_1'$ yields the goal: $\Gamma_1 + r * \Gamma_2 \vdash [Ctxt'\{t\}] : A'$

- $Ctxt = \mathbf{let}\,[y] = Ctxt'\ \mathbf{in}\ t_2$ where $Ctxt\{x\} = \mathbf{let}\ [y]\ = Ctxt'\{x\}\ \mathbf{in}\ t_2$. i.e., with premise derivation:

$$\dfrac{\Gamma_1', x : A \vdash Ctxt'\{x\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{(\Gamma_1' + \Gamma_2'), x : A \vdash \mathbf{let}\ [y]\ = Ctxt'\{x\}\ \mathbf{in}\ t_2 : B}\ \text{ELIM}$$

Proceeds by induction similarly to the (App-L) case, with no extra context manipulation necessary.

- $Ctxt = \textbf{let unit} = Ctxt' \textbf{ in } t_2$ where $Ctxt\{x\} = \textbf{let unit} = Ctxt'\{x\} \textbf{ in } t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : 1 \qquad \Gamma_2' \vdash t_2 : B}{(\Gamma_1' + \Gamma_2'), x : A \vdash \textbf{let } [y] \ = Ctxt'\{x\} \textbf{ in } t_2 : B} \text{ ELIM}$$

  Proceeds by induction similarly to the (App-L) case, with no extra context manipulation necessary.
- $Ctxt = \textbf{let } (y, z) = Ctxt' \textbf{ in } t_2$ where $Ctxt\{x\} = \textbf{let } (y, z) = Ctxt'\{x\} \textbf{ in } t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : A_1 \otimes A_2 \qquad \Gamma_2', y : A_1, z : A_2 \vdash t_2 : B}{(\Gamma_1' + \Gamma_2'), x : A \vdash \textbf{let } (y, z) = Ctxt'\{x\} \textbf{ in } t_2 : B} \text{ ELIM}$$

  Proceeds by induction similarly to the (App-L) case, with no extra context manipulation necessary.
- $Ctxt = (Ctxt', t')$ where $Ctxt\{x\} = (Ctxt'\{x\}, t')$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : A' \qquad \Gamma_2' \vdash t' : B'}{(\Gamma_1' + \Gamma_2'), x : A \vdash (Ctxt'\{x\}, t') : A' \otimes B'} \otimes_i$$

  Proceeds by induction similarly to the (App-L) case, with no extra context manipulation necessary.
- $Ctxt = (v, Ctxt')$ where $Ctxt\{x\} = (v, Ctxt'\{x\})$. i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash v : A' \qquad \Gamma_2', x : A \vdash Ctxt'\{x\} : B'}{(\Gamma_1' + \Gamma_2'), x : A \vdash (v, Ctxt'\{x\}) : A' \otimes B'} \otimes_i$$

  Proceeds by induction similarly to the (App-R) case, with no extra context manipulation necessary. □

Lemma 9. *Value lemma*
Given $\Gamma \vdash v : A$ then, depending on the type, the shape of v can be inferred:

- *$A = A' \multimap B$ then $v = \lambda x.v'$ or a partially applied primitive term p.*
- *$A = \Box_r A'$ then $v = [v']$.*
- *$A = A' \otimes B$ then $v = (v_1, v_2)$.*
- *$A = 1$ then $v = \textbf{unit}$.*
- *$A = \textbf{Chan } P$ then $v = c$.*

Proof. Recall that the value terms sub-grammar is:

$$v ::= (v_1, v_2) \mid \textbf{unit} \mid \ [v] \mid \lambda x.v \mid c \mid p \qquad \text{(value terms sub-grammar)}$$

where p are partially-applied primitives. In other words,

$$\begin{aligned} p ::= \ & \mathsf{close} \mid \mathsf{send} \mid \mathsf{send}\, c \mid \mathsf{receive} \mid \mathsf{forkLinear} \mid \mathsf{forkNonLinear} \\ & \mid \mathsf{forkReplicate} \mid \mathsf{forkReplicate}\, c \\ & \mid \mathsf{forkMulticast} \mid \mathsf{forkMulticast}\, c \end{aligned}$$

We then proceed by case analysis on the type A to match the structure of the lemma. In each case we must consider what possible values can be assigned the type A and by which rules.

In all cases, there exist additional derivations based on dereliction, weakening and approximation, e.g., for the case where $A = A' \multimap B$:

$$\frac{\Gamma, x : A' \vdash t : A' \multimap B}{\Gamma, x : \Box_1 A' \vdash t : A' \multimap B} \text{ DER}$$

$$\frac{\Gamma, x : A' \vdash t : A' \multimap B}{\Gamma, x : \Box_1 A', 0 * \Delta \vdash t : A' \multimap B} \text{ DER}$$

$$\frac{\Gamma, y : \Box_r A'', \Gamma', x : A' \vdash t : A' \multimap B \quad r \sqsubseteq s}{\Gamma, y : \Box_s A'', \Gamma', x : A' \vdash t : A' \multimap B} \text{ DER}$$

In all of these cases we can apply induction on the premise to get the result since the term is preserved between the premise and the conclusion.

We elide handling this separately each time in the cases that follow as the reasoning through dereliction is the same each time.

- $A = A' \multimap B$ then there are two classes of possible typing:
  - Abstract term:

$$\frac{\Gamma, x : A' \vdash v' : B}{\Gamma \vdash \lambda x.v' : A' \multimap B} \multimap_i$$

    thus $v = \lambda x.v'$ as in the lemma statement.
  - Primitive term p formed by an application of zero or more values to a primitive operation, of which there are then ten possibilities:
    1.

$$\frac{}{\emptyset \vdash \mathsf{close} : \mathbf{Chan}\,\mathbf{End} \multimap 1}\ \textsc{close}$$

    then $v = \mathsf{close}$.
    2.

$$\frac{}{\emptyset \vdash \mathsf{send} : A \otimes \mathbf{Chan}\,\mathbf{Send}\,A\,P \multimap \mathbf{Chan}\,P}\ \textsc{send}$$

    then $v = \mathsf{send}$.
    3.

$$\frac{\dfrac{}{c : \mathbf{Chan}\,P \vdash c : \mathbf{Chan}\,P}\ \textsc{var} \qquad \dfrac{}{\emptyset \vdash \mathsf{send} : A \otimes \mathbf{Chan}\,\mathbf{Send}\,A\,P \multimap \mathbf{Chan}\,P}\ \textsc{send}}{c : \mathbf{Chan}\,P \vdash \mathsf{send}\,c : A \multimap \mathbf{Chan}\,P} \multimap_e$$

    then $v = \mathsf{send}\,c$.
    4.

$$\frac{}{\emptyset \vdash \mathsf{receive} : \mathbf{Chan}\,\mathbf{Recv}\,A\,P \multimap A \otimes \mathbf{Chan}\,P}\ \textsc{recv}$$

    then $v = \mathsf{receive}$.
    5.

$$\frac{}{\emptyset \vdash \mathsf{forkLinear} : (\mathbf{Chan}\,P \multimap 1) \multimap \mathbf{Chan}\,\overline{P}}\ \textsc{fork}$$

    then $v = \mathsf{forkLinear}$.
    6.

$$\frac{}{\emptyset \vdash \mathsf{forkNonLinear} : (\square_r \mathbf{Chan}\,P \multimap 1) \multimap \square_r(\mathbf{Chan}\,\overline{P})}\ \textsc{forkNonLin}$$

    then $v = \mathsf{forkNonLinear}$.
    7.

$$\frac{\mathsf{ReceivePrefix}(P)}{\emptyset \vdash \mathsf{forkReplicate} : \square_{0..n}(\mathbf{Chan}\,P \multimap 1) \multimap \mathbf{N}\,n \multimap \mathbf{Vec}\,n\,(\square_{0..1}(\mathbf{Chan}\,\overline{P}))}\ \textsc{forkRep}$$

    then $v = \mathsf{forkReplicate}$.
    8.

$$\frac{\dfrac{\dfrac{}{c : \mathbf{Chan}\,P \vdash c : \mathbf{Chan}\,P}\ \textsc{var}}{c : \square_{0..1}\mathbf{Chan}\,P \vdash [c] : \square_{0..1}\mathbf{Chan}\,P}\ \textsc{pr} \qquad \dfrac{\mathsf{ReceivePrefix}(P)}{\emptyset \vdash \mathsf{forkReplicate} : \square_{0..n}(\mathbf{Chan}\,P \multimap 1) \multimap \mathbf{N}\,n \multimap \mathbf{Vec}\,n\,(\square_{0..1}(\mathbf{Chan}\,\overline{P}))}\ \textsc{forkRep}}{c : \square_{0..1}\mathbf{Chan}\,P \vdash \mathsf{forkReplicate}\,[c] : \mathbf{N}\,n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})} \multimap_e$$

    then $v = \mathsf{forkReplicate}\,[c]$.
    9.

$$\frac{\mathsf{Sends}(P)}{\emptyset \vdash \mathsf{forkMulticast} : (\mathbf{Chan}\,(\mathsf{Graded}_n P) \multimap 1) \multimap \mathbf{N}\,n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})}\ \textsc{forkMulti}$$

    then $v = \mathsf{forkMulticast}$.

10.

$$\dfrac{\dfrac{}{c : \mathbf{Chan}\,(\mathsf{Graded}_n P) \vdash c : \mathbf{Chan}\,(\mathsf{Graded}_n P)}\,\textsc{var} \qquad \dfrac{\mathsf{Sends}(P)}{\emptyset \vdash \mathsf{forkMulticast} : (\mathbf{Chan}\,(\mathsf{Graded}_n P) \multimap 1) \multimap \mathbf{N}\; n \multimap \mathbf{Vec}\; n\; (\mathbf{Chan}\,\overline{P})}\,\textsc{forkMulti}}{c : \mathbf{Chan}\,(\mathsf{Graded}_n P) \vdash \mathsf{forkMulticast}\, c : \mathbf{N}\; n \multimap \mathbf{Vec}\; n\; (\mathbf{Chan}\,\overline{P})} \multimap_e$$

then $v = \mathsf{forkMulticast}\, c$.

- $A = \Box_r A'$ then there is only one possible non-dereliction typing of a value at that type:

$$\dfrac{\Gamma \vdash v' : A'}{r * \Gamma \vdash [v'] : \Box_r A'}\,\textsc{pr}$$

thus $v = [v']$ as in the lemma statement.

- $A = A' \otimes B$ then there is only one possible non-dereliction typing of a value at that type:

$$\dfrac{\Gamma_1 \vdash v_1 : A' \quad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : A' \otimes B}\,\otimes_i$$

thus $v = (v_1, v_2)$ as in the lemma statement.

- $A = 1$ then there is only one possible non-dereliction typing of a value at that type:

$$\dfrac{}{\emptyset \vdash \mathbf{unit} : 1}\,\textsc{unit}_i$$

thus $v = \mathbf{unit}$ as in the lemma statement.

- $A = \mathbf{Chan}\, P$ then the only possible typing that is a value is given by:

$$\dfrac{}{c : \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\,\textsc{var}$$

thus $v = c$ as in the lemma statement, regardless of the protocol P. □

Appendix C. Progress proof

Theorem 1 *(Progress terms).*

$$\begin{aligned} &\Delta \vdash t : A \\ &\wedge\ \Delta \vdash \mathcal{C} \\ &\wedge\ \mathsf{buffered}(\mathcal{C}, \Delta) \\ &\wedge\ \Delta \vdash \mathcal{P} : B \\ \implies\ &\mathsf{value}(t) \vee \exists t', \mathcal{C}', \mathcal{P}'.\ (\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t';\ \mathcal{C}') \end{aligned}$$

Proof. We prove this by induction on the three judgments forming the assumptions.
First, we case split on the typing of process configurations: $\Delta \vdash \mathcal{P} : B$.

- (cons)

$$\dfrac{\begin{array}{c}\Gamma_1, \Delta_1 \vdash \mathcal{P} : A \\ \Gamma_2, \Delta_2 \vdash t : B\end{array}}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash \mathcal{P} \mid t^{\mathcal{R}} : A}\,\textsc{cons}\mathcal{R}$$

In this case, we can write the process configuration $\mathcal{P}$ as $\mathcal{P}' \mid t''$, or in other words as another process configuration $\mathcal{P}'$ extended with some term t''.
By induction, there are two possibilities here: either t'' is itself a value, or there exists some reduction $(\mathcal{P}';\ \mathcal{C}) \rightsquigarrow (\mathcal{P}'';\ \mathcal{C}')$.
If there is such a reduction, then we can make a global reduction using the following rule:

$$\dfrac{(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')}{(\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t;\ \mathcal{C}')}\,\textsc{par}$$

and the overall progress theorem is satisfied.
Otherwise, we must consider the possible reductions for the additional term t.

- (empty)

$$\frac{}{\emptyset \vdash \cdot : \bot}\ \text{EMPTY}$$

Since the process configuration is empty in this case, it cannot make any reductions, so again we only need to consider the possible reductions originating from the term t.
We then induct on the last step of the typing derivation $\Delta_1 \vdash t : A$. Note that in some cases the process and channel configurations do not come into play, in which case the proof proceeds roughly the same as the standard progress theorem on terms; either t is a value, in which case the theorem holds directly by induction, or we can take a step $t \rightsquigarrow t'$, in which case we will be able to take a global step via the Reduce rule. In these cases we can generally ignore the presence of the runtime context Δ_1.

– (var)

$$\frac{}{x : A \vdash x : A}\ \text{VAR}$$

Trivial since the variable x cannot be reduced and so is necessarily a value.

– (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B} \multimap_i$$

Trivial since $(\lambda x.t)$ is already a value.

– (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B} \multimap_e$$

By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'.
In the first case, by the value lemma there are eleven cases to consider.

1. $t_1 = \lambda x.t_1'$. Here, there are two possibilities: either $t_2 = v$ for some value v or $t_2 \rightsquigarrow t_2'$ for some t_2'.
   In the first case, we can reduce by Reduce using the reduction:

$$\frac{}{(\lambda x.t)\, v \rightsquigarrow t[v/x]} \rightsquigarrow_\beta$$

   In the second case, since $\lambda x.t_1'$ is a value then by induction on the premise we can reduce by Reduce using the reduction:

$$\frac{t \rightsquigarrow t'}{v\, t \rightsquigarrow v\, t'} \rightsquigarrow_{\text{prim}}$$

2. $t_1 = \mathsf{close}$, where $A = \mathbf{Chan\ End}$. Therefore, we induct on the second argument. There are two cases.
   * t_2 is a value. Then, by the value lemma, $t_2 = c$ and the runtime typing derivation has two possible (consequential) specialisations:

   (a)

$$\frac{\dfrac{}{c : \mathbf{Chan\ End} \vdash c : \mathbf{Chan\ End}}\ \text{VAR} \qquad \dfrac{}{\emptyset \vdash \mathsf{close} : \mathbf{Chan\ End} \multimap 1}\ \text{CLOSE}}{c : \mathbf{Chan\ End} \vdash \mathsf{close}\, c : 1} \multimap_e$$

   Therefore, by Lemma 7 our overall succedent is of the form $\Delta_2', c : \mathbf{Chan\ End} \vdash Ctxt\{\mathsf{close}\, c\} : A$.
   By inversion on the typing of channel configurations, we necessarily have:

$$\frac{\Delta \vdash \mathcal{C}}{\Delta, c :^r \mathbf{Chan\ End} \vdash \mathcal{C}, c = \langle\rangle}\ \text{EMPTY}$$

   Therefore, there is a reduction as follows:

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{close}\, c\};\ \mathcal{C}, c^1 = \langle\rangle) \rightsquigarrow (\mathcal{P} \mid Ctxt\{\mathbf{unit}\};\ \mathcal{C})}\ \text{CLOSE}$$

(b)

$$\dfrac{\dfrac{}{c :^* \mathbf{Chan}\,\mathbf{End} \vdash c : \mathbf{Chan}\,\mathbf{End}}\,\text{VARS} \quad \dfrac{}{\emptyset \vdash \mathsf{close} : \mathbf{Chan}\,\mathbf{End} \multimap 1}\,\text{CLOSE}}{c :^* \mathbf{Chan}\,\mathbf{End} \vdash \mathsf{close}\,c : 1} \multimap_e$$

Therefore, by Lemma 7 our overall succedent is of the form $\Delta_2', c :^* \mathbf{Chan}\,\mathbf{End} \vdash \mathit{Ctxt}\{\mathsf{close}\,c\} : A$.
There is then a reduction as follows:

$$\dfrac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{close}\,c\};\ \mathcal{C}, c^* = \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{\mathbf{unit}\};\ \mathcal{C}, c^* = \vec{v})}\,\text{SHAREDCLOSE}$$

* t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by REDUCE using the reduction:

$$\dfrac{t_2 \rightsquigarrow t_2'}{\mathsf{close}\,t_2 \rightsquigarrow \mathsf{close}\,t_2'}\,\rightsquigarrow_{\mathsf{prim}}$$

3. $t_1 = \mathsf{send}$, where $A = \mathbf{Chan}\,(\mathbf{Send}\,A'\,P) \multimap A' \multimap \mathbf{Chan}\,P$. Therefore, we induct on the second argument. There are two cases.
   * t_2 is a value. Then, by the value lemma, $t_2 = c$ and so $t_1\,t_2 = \mathsf{send}\,c$ which is also a value.
   * t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by REDUCE using the reduction:

$$\dfrac{t_2 \rightsquigarrow t_2'}{\mathsf{send}\,t_2 \rightsquigarrow \mathsf{send}\,t_2'}\,\rightsquigarrow_{\mathsf{prim}}$$

4. $t_1 = \mathsf{send}\,c$, where $A = A' \multimap \mathbf{Chan}\,P$. Therefore, we induct on the second argument. There are two cases.
   * $t_2 = x$ for some value x. The runtime typing derivation specialises to:

$$\dfrac{\dfrac{\cdots}{c :^r \mathbf{Chan}\,(\mathbf{Send}\,A'\,P) \vdash \mathsf{send}\,c : A' \multimap \mathbf{Chan}\,P} \multimap_e \quad \dfrac{}{\emptyset \vdash v : A'}\,\text{VAR}}{c :^r \mathbf{Chan}\,(\mathbf{Send}\,A'\,P) \vdash \mathsf{send}\,c\,x : \mathbf{Chan}\,P} \multimap_e$$

Therefore, by Lemma 7 our overall succedent is of the form $\Delta_2', c :^r \mathbf{Chan}\,(\mathbf{Send}\,A'\,P) \vdash \mathit{Ctxt}\{\mathsf{send}\,c\,x\} : A$.
By inversion, channel configuration typing must be of the form:

$$\dfrac{\begin{array}{l}\Delta_1 \vdash v : A \\ \Delta_1, \Delta_2, c :^r \mathbf{Chan}\,P \vdash \mathcal{C}, \overline{c} = \vec{v} : v\end{array}}{\Delta_1, \Delta_2, c :^r \mathbf{Chan}\,(\mathbf{Send}\,A\,P) \vdash \mathcal{C}, \overline{c} = \vec{v}}\,\text{SEND}$$

with $\Delta = \Delta_1, \Delta_2$.
In this case, there is then a reduction as follows:

$$\dfrac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{send}\,c\,v\};\ \mathcal{C}, \overline{c} = \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{c\};\ \mathcal{C}, \overline{c} = \vec{v} : v)}\,\text{SEND}$$

   * t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by REDUCE using the reduction:

$$\dfrac{t_2 \rightsquigarrow t_2'}{\mathsf{send}\,c\,t_2 \rightsquigarrow \mathsf{send}\,c\,t_2'}\,\rightsquigarrow_{\mathsf{prim}}$$

5. $t_1 = \mathsf{receive}$, where $A = \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \multimap A' \otimes \mathbf{Chan}\,P$. Therefore, we induct on the second argument. There are two cases.
   * t_2 is a value. Then, by the value lemma, $v_2 = c$ and the runtime typing derivation refines to:

$$\dfrac{\dfrac{\cdots}{c :^r \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \vdash c : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P)}\,\text{VAR} \quad \dfrac{}{\emptyset \vdash \mathsf{receive} : \mathbf{Chan}\,\mathbf{Recv}\,A\,P \multimap A \otimes \mathbf{Chan}\,P}\,\text{RECV}}{c :^r \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \vdash \mathsf{receive}\,c : A' \otimes \mathbf{Chan}\,P} \multimap_e$$

Therefore, by Lemma 7 our overall succedent is of the form $\Delta_2', c :^r \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \vdash \mathit{Ctxt}\{\mathsf{receive}\,c\} : A$.
Now, since the channel configuration $\mathcal{C}$ must be buffered with respect to Δ, the channel c cannot be empty, and so it must contain at least one value v. Hence, there is only one possibility for the channel configuration typing:

$$\dfrac{\begin{array}{l}\Delta_1 \vdash v : A \\ \Delta_1, \Delta_2, c :^r \mathbf{Chan}\,P \vdash \mathcal{C}, c = \vec{v}\end{array}}{\Delta_1, \Delta_2, c :^r \mathbf{Chan}\,(\mathbf{Recv}\,A\,P) \vdash \mathcal{C}, c = v : \vec{v}}\,\text{RECV}$$

with $\Delta = \Delta_1, \Delta_2$.
Then there is a reduction as follows:

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{receive}\, c\};\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{(v, c)\};\ \mathcal{C}, c = \vec{v})}\ \textsc{recv}$$

* t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by Reduce using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{receive}\, t_2 \rightsquigarrow \mathsf{receive}\, t_2'}\ \rightsquigarrow_{\mathsf{prim}}$$

6. $t_1 = \mathsf{forkLinear}$, where $A = (\mathbf{Chan}\, P \multimap 1) \multimap \mathbf{Chan}\, \overline{P}$. Therefore, we induct on the second argument. There are two cases.
   * t_2 is a value. Then, by the value lemma, $t_2 = \lambda x.t_2'$ for some t_2' and we can construct the following runtime typing derivation:

$$\frac{\dfrac{x : \mathbf{Chan}\, P \vdash t_2' : 1}{\emptyset \vdash \lambda x.t_2' : \mathbf{Chan}\, P \multimap 1}\ \multimap_i \qquad \dfrac{}{\emptyset \vdash \mathsf{forkLinear} : (\mathbf{Chan}\, P \multimap 1) \multimap \mathbf{Chan}\, \overline{P}}\ \textsc{fork}}{\emptyset \vdash \mathsf{forkLinear}\, \lambda x.t_2' : \mathbf{Chan}\, \overline{P}}\ \multimap_e$$

   Therefore, by Lemma 7 our overall succedent is of the form $\Delta_2' \vdash Ctxt\{\mathsf{forkLinear}\, \lambda x.t_2'\} : A$.
   Then there is a reduction as follows:

$$\frac{\#c}{(\mathcal{P} \mid Ctxt\{\mathsf{forkLinear}\, (\lambda x.t)\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^\circ \mid Ctxt\{\overline{c}\};\ \mathcal{C}, \overline{c} = \langle\rangle, c = \langle\rangle)}\ \textsc{forkLinear}$$

   where the channel configuration is typed $\Delta_2' = \mathcal{C}$ and we select a fresh name c for the channel being forked.
   * t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by Reduce using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{forkLinear}\, t_2 \rightsquigarrow \mathsf{forkLinear}\, t_2'}\ \rightsquigarrow_{\mathsf{prim}}$$

7. $t_1 = \mathsf{forkNonLinear}$, where $A = (\Box_r(\mathbf{Chan}\, P) \multimap 1) \multimap \Box_r(\mathbf{Chan}\, \overline{P})$. Therefore, we induct on the second argument. There are two cases.
   * t_2 is a value. Then, by the value lemma, $t_2 = \lambda x.t_2'$ for some t_2' and we can construct the following runtime typing derivation:

$$\frac{\dfrac{x : \Box_r(\mathbf{Chan}\, P) \vdash t_2' : 1}{\emptyset \vdash \lambda x.t_2' : \Box_r(\mathbf{Chan}\, P) \multimap 1}\ \multimap_i \qquad \dfrac{}{\emptyset \vdash \mathsf{forkNonLinear} : (\Box_r \mathbf{Chan}\, P \multimap 1) \multimap \Box_r(\mathbf{Chan}\, \overline{P})}\ \textsc{forkNonLin}}{\emptyset \vdash \mathsf{forkNonLinear}\, \lambda x.t_2' : \Box_r(\mathbf{Chan}\, \overline{P})}\ \multimap_e \tag{C.1}$$

   Note that this derivation cannot include any instances of the Approx rule, because of the `ExactSemiring` constraint present in the type signature of `forkNonLinear`, and so the grade r must remain the same throughout the typing.
   Therefore, by Lemma 7 our overall succedent is of the form $\Delta_2' \vdash Ctxt\{\mathsf{forkNonLinear}\, \lambda x.t_2'\} : A$.
   Then there is a reduction as follows:

$$\frac{\#c}{(\mathcal{P} \mid Ctxt\{\mathsf{forkNonLinear}\, (\lambda x.t)\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{[\overline{c}]\} \mid t[[c]/x]^\circ;\ \mathcal{C}, \overline{c}^* = \langle\rangle, c^* = \langle\rangle)}\ \textsc{forkNonLinear}$$

   where the channel configuration is typed $\Delta_2' = \mathcal{C}$ and we select a fresh name c for the channel being forked.
   * t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by Reduce using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{forkNonLinear}\, t_2 \rightsquigarrow \mathsf{forkNonLinear}\, t_2'}\ \rightsquigarrow_{\mathsf{prim}}$$

8. $t_1 = \mathsf{forkReplicate}$, where $A = \Box_{0..1}(\mathbf{Chan}\, P \multimap 1) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})$. Therefore, we induct on the second argument. There are two cases.
   * t_2 is a value. Then, by the value lemma, $t_2 = [\lambda c.t_2']$ for some t_2' and $t_1\, t_2$ is $\mathsf{forkReplicate}\, [\lambda c.t_2']$ which is also a value.

∗ t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by Reduce using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{forkReplicate}\, t_2 \rightsquigarrow \mathsf{forkReplicate}\, t_2'} \rightsquigarrow_{\mathsf{prim}}$$

9. $t_1 = \mathsf{forkReplicate}\, c$, where $A = \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})$. Therefore, we induct on the second argument. There are two cases.
   ∗ t_2 is a value. Then, by the value lemma, $t_2 = \lambda x.t_2'$ for some t_2' and the runtime typing derivation refines to:

$$\dfrac{\dfrac{\dfrac{x : \mathbf{Chan}\, P \vdash t_2' : 1}{\emptyset \vdash \lambda x.t_2' : \mathbf{Chan}\, P \multimap 1} \multimap_i}{\emptyset \vdash [\lambda x.t_2'] : \square_{0..n}(\mathbf{Chan}\, P \multimap 1)}\ \mathrm{PR} \quad \dfrac{\mathsf{ReceivePrefix}(P)}{\emptyset \vdash \mathsf{forkReplicate} : \square_{0..n}(\mathbf{Chan}\, P \multimap 1) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\square_{0..1}(\mathbf{Chan}\, \overline{P}))}\ \textsc{forkRep}}{\emptyset \vdash \mathsf{forkReplicate}\, [\lambda x.t_2'] : \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\square_{0..1}(\mathbf{Chan}\, \overline{P}))} \multimap_e}{\emptyset \vdash \mathsf{forkReplicate}\, [\lambda x.t_2']\, n : \mathbf{Vec}\, n\, (\square_{0..1}(\mathbf{Chan}\, \overline{P}))} \multimap_e$$

   Therefore, by Lemma 7 our overall succedent is of the form $\Delta_2' \vdash \mathit{Ctxt}\{\mathsf{forkReplicate}\, [\lambda x.t_2']\, n\} : A$.
   Then there is a reduction as follows:

$$\frac{\#c_1 \ldots \#c_n}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkReplicate}\, [\lambda x.t]\, n\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{\langle [\overline{c_1}], .., [\overline{c_n}] \rangle\} \mid \mathsf{Wch}\, (c_1 \mapsto \lambda x.t) \mid .. \mid \mathsf{Wch}\, (c_n \mapsto \lambda x.t);\ \mathcal{C}, c_1 = \langle\rangle, .., c_n = \langle\rangle, \overline{c_1} = \langle\rangle, .., \overline{c_n} = \langle\rangle)}\ \textsc{forkReplicate}$$

   where the channel configuration is typed $\Delta_2' = \mathcal{C}$ and we select fresh names c and $c_1 \ldots c_n$ for the channels being forked.
   ∗ t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by Reduce using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{forkReplicate}\, c\, t_2 \rightsquigarrow \mathsf{forkReplicate}\, c\, t_2'} \rightsquigarrow_{\mathsf{prim}}$$

10. $t_1 = \mathsf{forkMulticast}$, where $A = (\mathbf{Chan}\, (\mathsf{Graded}_n P) \multimap 1) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})$. Therefore, we induct on the second argument. There are two cases.
   ∗ t_2 is a value. Then, by the value lemma, $t_2 = \lambda c.t_2'$ for some t_2' and $t_1\, t_2$ is $\mathsf{forkMulticast}\, \lambda c.t_2'$ which is also a value.
   ∗ t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by Reduce using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{forkMulticast}\, t_2 \rightsquigarrow \mathsf{forkMulticast}\, t_2'} \rightsquigarrow_{\mathsf{prim}}$$

11. $t_1 = \mathsf{forkMulticast}\, c$, where $A = \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})$. Therefore, we induct on the second argument. There are two cases.
   ∗ t_2 is a value. Then, by the value lemma, $t_2 = \lambda x.t_2'$ for some t_2' and the runtime typing derivation refines to:

$$\dfrac{\dfrac{\dfrac{x : \mathbf{Chan}\, (\mathsf{Graded}_r P) \vdash t_2' : 1}{\emptyset \vdash \lambda x.t_2' : \mathbf{Chan}\, (\mathsf{Graded}_r P) \multimap 1} \multimap_i}{\emptyset \vdash \mathsf{forkMulticast}\, (\lambda x.t_2') : \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})} \multimap_e \quad \dfrac{\mathsf{Sends}(P)}{\emptyset \vdash \mathsf{forkMulticast} : (\mathbf{Chan}\, (\mathsf{Graded}_n P) \multimap 1) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})}\ \textsc{forkMulti}}{\emptyset \vdash \mathsf{forkMulticast}\, (\lambda x.t_2')\, n : \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})} \multimap_e$$

   Therefore, by Lemma 7 our overall succedent is of the form $\Delta_2' \vdash \mathit{Ctxt}\{\mathsf{forkMulticast}\, \lambda x.t_2'\, n\} : A$.
   Then there is a reduction as follows:

$$\frac{\#c \quad \#c_1 \ldots \#c_n}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkMulticast}\, (\lambda x.t)\, n\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{\langle \overline{c_1}, .., \overline{c_n} \rangle\} \mid t[c/x]^\circ \mid \mathsf{Fwd}\, (c \mapsto c_1, .., c_n);\ \mathcal{C}, c = \langle\rangle, \overline{c} = \langle\rangle, c_1 = \langle\rangle, .., c_n = \langle\rangle, \overline{c_1} = \langle\rangle, .., \overline{c_n} = \langle\rangle)}\ \textsc{forkMulticast}$$

   where the channel configuration is typed $\Delta_2' = \mathcal{C}$ and we select fresh names c and $c_1 \ldots c_n$ for the channels being forked.
   ∗ t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by Reduce using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{forkMulticast}\, c\, t_2 \rightsquigarrow \mathsf{forkMulticast}\, c\, t_2'} \rightsquigarrow_{\mathsf{prim}}$$

In the second case, by induction on the premise we can reduce by Reduce using the reduction:

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_2} \rightsquigarrow_{\mathsf{APP}}$$

– (pr)

$$\frac{\Gamma \vdash t : A}{r * \Gamma \vdash [t] : \Box_r A}\,\text{PR}$$

There are two cases: either $t = v$ for some value v, or $t \rightsquigarrow t'$ for some t'.
In the first case, $[v]$ is a value.
In the second case, we can reduce by Reduce using the reduction:

$$\frac{t \rightsquigarrow t'}{[t] \rightsquigarrow [t']}\rightsquigarrow\Box\text{reduce}$$

– (elim)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ [x]\ = t_1\ \textbf{in}\ t_2 : B}\,\text{ELIM}$$

By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'.
In the first case, by the value lemma $t_1 = [v']$ for some value v'. This refines the typing as follows:

$$\frac{\dfrac{\emptyset \vdash v' : A}{\emptyset \vdash [v'] : \Box_r A}\,\text{PR} \qquad x : [A]_r \vdash t_2 : B}{\emptyset \vdash \textbf{let}\ [x]\ = [v']\ \textbf{in}\ t_2 : B}\,\text{ELIM}$$

Therefore, we can reduce by Reduce using the reduction:

$$\frac{}{\textbf{let}\ [x]\ = [v]\ \textbf{in}\ t \rightsquigarrow t[v/x]}\rightsquigarrow\Box\beta$$

In the second case, by induction on the premise we can reduce by Reduce using the reduction:

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let}\ [x]\ = t_1\ \textbf{in}\ t_2 \rightsquigarrow \textbf{let}\ [x]\ = t_1'\ \textbf{in}\ t_2}\rightsquigarrow_{\text{LET}\Box}$$

– (der)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\,\text{DER}$$

Goal achieved immediately by induction on the premise.

– (weak)

$$\frac{\Gamma \vdash t : B}{\Gamma, 0 * \Gamma' \vdash t : B}\,\text{WEAK}$$

Goal achieved immediately by induction on the premise.

– (approx)

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\,\text{APPROX}$$

Goal achieved immediately by induction on the premise.

– (tensor)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\otimes_i$$

There are three cases: either $t_1 = v_1$ and $t_2 = v_2$ for some pair of values (v_1, v_2), $t_1 = v$ for some value v but $t_2 \rightsquigarrow t_2'$ for some t_2', or $t_1 \rightsquigarrow t_1'$ for some t_1'.
In the first case, (v_1, v_2) is a value.
In the second case, we can reduce by Reduce using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{(v, t_2) \rightsquigarrow (v, t_2')}\rightsquigarrow\otimes\text{reduceR}$$

In the third case, we can reduce by Reduce using the reduction:

$$\frac{t_1 \rightsquigarrow t_1'}{(t_1, t_2) \rightsquigarrow (t_1', t_2)} \rightsquigarrow_{\otimes \text{reduceL}}$$

– (pairElim)

$$\frac{\begin{array}{c}\Gamma_1 \vdash t_1 : A \otimes A' \\ \Gamma_2, x : A, y : A' \vdash t_2 : B\end{array}}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ (x, y) = t_1\ \textbf{in}\ t_2 : B} \otimes_e$$

By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'. In the first case, by the value lemma $t_1 = (v_1, v_2)$ for some pair of values (v_1, v_2). This refines the typing as follows:

$$\frac{\dfrac{\emptyset \vdash v_1 : A \quad \emptyset \vdash v_2 : A'}{\emptyset \vdash (v_1, v_2) : A \otimes A'} \otimes_i \qquad x : A, y : A' \vdash t_2 : B}{\emptyset \vdash \textbf{let}\ (x, y) = (v_1, v_2)\ \textbf{in}\ t_2 : B} \otimes_e$$

Therefore, we can reduce by Reduce using the reduction:

$$\frac{}{\textbf{let}\ (x, y) = (v_1, v_2)\ \textbf{in}\ t_3 \rightsquigarrow t_3[v_1/x][v_2/y]} \rightsquigarrow_{\otimes\beta}$$

In the second case, by induction on the premise we can reduce by Reduce using the reduction:

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let}\ (x, y) = t_1\ \textbf{in}\ t_2 \rightsquigarrow \textbf{let}\ (x, y) = t_1'\ \textbf{in}\ t_2} \rightsquigarrow_{\text{LET}\otimes}$$

– (unit)

$$\frac{}{\emptyset \vdash \textbf{unit} : 1}\ \text{UNIT}_i$$

A value.

– (unitElim)

$$\frac{\begin{array}{c}\Gamma_1 \vdash t_1 : 1 \\ \Gamma_2 \vdash t_2 : B\end{array}}{\Gamma_1 + \Gamma_2 \vdash \textbf{let unit} = t_1\ \textbf{in}\ t_2 : B}\ \text{UNIT}_e$$

By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'. In the first case, by the value lemma $t_1 = \textbf{unit}$ and so we can reduce by Reduce using the reduction:

$$\frac{}{\textbf{let unit} = \textbf{unit in}\ t \rightsquigarrow t} \rightsquigarrow_{\beta\text{unit}}$$

In the second case, by induction on the premise we can reduce by Reduce using the reduction:

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let unit} = t_1\ \textbf{in}\ t_2 \rightsquigarrow \textbf{let unit} = t_1'\ \textbf{in}\ t_2} \rightsquigarrow_{\text{LET}\text{unit}} \quad \square$$

A process configuration $\mathcal{P}$ contains all values, defined by the predicate $\mathsf{values}(\mathcal{P})$, if every process is either a value v or is a forwarder or watcher process.

Theorem 2 *(Progress global config).*

$$\begin{array}{l}\Delta \vdash \mathcal{P} : A \ \wedge\ \Delta \vdash \mathcal{C}\ \wedge\ \mathsf{buffered}(\mathcal{C}, \Delta) \\ \Longrightarrow\ \mathsf{values}(\mathcal{P}) \vee \exists \mathcal{C}', \mathcal{P}'.(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')\end{array}$$

Proof. By induction on the two judgments forming the assumptions.

First, we case split on the typing of process configurations: $\Delta \vdash \mathcal{P} : A$.

- (empty)

$$\frac{}{\emptyset \vdash \cdot : \bot}\ \text{EMPTY}$$

This case is trivial, as $\mathsf{values}(\mathcal{P})$ holds by default for an empty process configuration.

- (cons)

$$\frac{\Gamma_1, \Delta_1 \vdash \mathcal{P} : A \qquad \Gamma_2, \Delta_2 \vdash t : B}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash \mathcal{P} \mid t^{\mathcal{R}} : A}\ \text{CONS}\mathcal{R}$$

Here, we can decompose $\mathcal{P}$ into $\mathcal{P}' \mid t$. First, focus on the particular term t and apply Theorem 1.
Then, either t is a value or there exists some possible reduction $(\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t';\ \mathcal{C}')$.
If we can make such a reduction, then the theorem is satisfied. If not, then apply induction to the smaller process configuration $\mathcal{P}'$. We find that either $\mathsf{values}(\mathcal{P}')$ holds or there exists a reduction $(\mathcal{P}';\ \mathcal{C}) \rightsquigarrow (\mathcal{P}'';\ \mathcal{C}')$.
If $\mathsf{values}(\mathcal{P}')$, then since t is also a value we have $\mathsf{values}(\mathcal{P})$ and we are done.
Otherwise, we apply the following rule to find a reduction satisfying the theorem:

$$\frac{(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')}{(\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t;\ \mathcal{C}')}\ \text{PAR}$$

- (watch)

$$\frac{\Gamma_1, \Delta_1 \vdash \mathcal{P} : A \qquad \Gamma_2, x : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P), \Delta_2 \vdash t : 1}{\Gamma_1 + (0..1 * \Gamma_2), \Delta_1 + \Delta_2, c : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \vdash \mathcal{P} \mid \mathsf{Wch}\,(c \mapsto \lambda x.t) : A}\ \text{WATCH}$$

By induction on the first assumption, we have either that $\mathsf{values}(\mathcal{P})$ or that there exists some reduction $(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')$.
If there is such a reduction, we can make a global reduction as follows:

$$\frac{(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')}{(\mathcal{P} \mid \mathsf{Wch}\,(c \mapsto \lambda x.t);\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid \mathsf{Wch}\,(c \mapsto \lambda x.t);\ \mathcal{C}')}\ \text{PAR}$$

Otherwise, we must reduce on the watcher process itself. Note that due to the session typing of the channel in the runtime context, and the condition requiring that $\mathcal{C}$ is buffered, the channel configuration must be typed as follows:

$$\frac{\Delta_1 + \Delta_2 \vdash \mathcal{C} \qquad \Delta_1 + \Delta_2 \vdash v : A'}{\Delta_1 + \Delta_2, c : (\mathbf{Recv}\,A'\,P) \vdash \mathcal{C}, c = v : \vec{v}}\ \text{RECV} \tag{C.2}$$

Hence, we can construct the following global reduction:

$$\frac{}{(\mathcal{P} \mid \mathsf{Wch}\,(c \mapsto \lambda x.t);\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^{\circ};\ \mathcal{C}, c = v : \vec{v})}\ \text{WATCHER}$$

- (forward)

$$\frac{\mathsf{Sends}(P) \qquad \Gamma, \Delta \vdash \mathcal{P} : A}{\Gamma, \Delta, c : \mathbf{Chan}\,(\mathsf{Graded}_n\overline{P}), c_i : \mathbf{Chan}\,P \vdash \mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n) : A}\ \text{FWD}$$

By induction on the first assumption, we have either that $\mathsf{values}(\mathcal{P})$ or that there exists some reduction $(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')$.
If there is such a reduction, we can make a global reduction as follows:

$$\frac{(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')}{(\mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n);\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n);\ \mathcal{C}')}\ \text{PAR}$$

Otherwise, we must reduce on the forwarder process itself. Note that due to the session typing of the channel in the runtime context, and the condition requiring that $\mathcal{C}$ is buffered, then the channel configuration must be typed as follows:

$$\frac{\dfrac{\emptyset \vdash v : A'}{\emptyset \vdash [v] : \Box_n A'}\ \text{PR} \qquad \dfrac{\Delta, c_1 : \mathbf{Chan}\,P, .., c_n : \mathbf{Chan}\,P, c : \mathbf{Chan}\,\overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v, c = \vec{v}}{\Delta, c_1 : \mathbf{Chan}\,(\mathbf{Send}\,A\,P), .., c_n : \mathbf{Chan}\,(\mathbf{Send}\,A\,P), c : \mathbf{Chan}\,\overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n, c = \vec{v}}\ \text{SEND}^n}{\Delta, c_1 : \mathbf{Chan}\,(\mathbf{Send}\,A\,P), .., c_n : \mathbf{Chan}\,(\mathbf{Send}\,A\,P), c : \mathbf{Chan}\,(\mathbf{Recv}\,(\Box_n A')\,\overline{P}) \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n, c = [v] : \vec{v}}\ \text{RECV} \tag{C.3}$$

Hence, we can construct the following global reduction:

$$\frac{}{(\mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n);\ \mathcal{C}, c = [v] : \vec{v}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n) \rightsquigarrow (\mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n);\ \mathcal{C}, c = \vec{v}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v)}\ \text{FORWARDER}$$ □

Appendix D. Preservation proof

Theorem 3 *(Preservation).*

$$\begin{aligned}&\Gamma, \Delta \vdash \mathcal{P} : A \ \wedge \ \Delta \vdash \mathcal{C} \ \wedge \ (\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')\\ \Longrightarrow\ &\exists \Delta'.\ \Gamma, \Delta' \vdash \mathcal{P}' : A \ \wedge \ \Delta' \vdash \mathcal{C}'\end{aligned}$$

Proof. By induction on the three judgments forming the assumptions here.
We first case split on the typing of process configurations: $\Gamma, \Delta \vdash \mathcal{P} : A$.

- (empty)

$$\frac{}{\emptyset \vdash \cdot : \bot}\ \textsc{empty}$$

There is no reduction on the empty process configuration so this case is trivially true since the antecedent is false.
- (cons) We consider an abstraction over the two possibilities for extending a process configuration with a term which is either main or subordinate process:

$$\frac{\Gamma_1, \Delta_1 \vdash t : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash \mathcal{P} \mid t^{\mathcal{R}} : A \diamond^{\mathcal{R}} B}\ \textsc{cons}\mathcal{R}$$

where $\mathcal{R}$ abstracts over whether t is a subprocess $\circ$ or the main process $\bullet$ and $A \diamond^{\mathcal{R}} B$ selects the appropriate type depending on $\mathcal{R}$, i.e.,

$$\begin{aligned}&A \diamond^{\bullet} B = B \ \wedge \ A = \bot\\ &A \diamond^{\circ} B = A \ \wedge \ B = 1\end{aligned}$$

We then case on the reduction of process configurations consider the role $\mathcal{R}$ where pertinent.
– (send)

$$\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{send}\, c\, v\};\ \mathcal{C}, \overline{c} = \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{c\};\ \mathcal{C}, \overline{c} = \vec{v} : v)}\ \textsc{send}$$

with process configuration typing:

$$\frac{\Gamma_1, \Delta_1, c :^r \mathbf{Chan}\,(\mathbf{Send}\, A\, P) \vdash \mathit{Ctxt}\{\mathsf{send}\, c\, v\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c :^r \mathbf{Chan}\,(\mathbf{Send}\, A\, P) \vdash \mathcal{P} \mid \mathit{Ctxt}\{\mathsf{send}\, c\, v\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R} \tag{D.1}$$

and channel configuration typing:

$$\frac{\Delta_3 \vdash v : A \qquad \Delta_3, \Delta_4, c :^r \mathbf{Chan}\, P \vdash \mathcal{C}, \overline{c} = \vec{v} : v}{\Delta_3, \Delta_4, c :^r \mathbf{Chan}\,(\mathbf{Send}\, A\, P) \vdash \mathcal{C}, \overline{c} = \vec{v}}\ \textsc{send} \tag{D.2}$$

where $(\Delta_3, \Delta_4) = \Delta_1 + \Delta_2$.
We can then construct the consequent channel configuration typing given by the premise of (D.2):

$$\Delta_1 + \Delta_2, c :^r \mathbf{Chan}\, P \vdash \mathcal{C}, \overline{c} = v : \vec{v}$$

From typing $\Gamma_1, \Delta_1 \vdash \mathit{Ctxt}\{\mathsf{send}\, c\, v\} : B$, and by Lemma 6 we have two cases:
1. Typing $\Gamma_1', \Delta_1' \vdash \mathsf{send}\, c\, v : A''$ and $\Gamma_2', y : A'', \Delta_2' \vdash \mathit{Ctxt}\{y\} : B$ with $\Gamma_1 = \Gamma_1' + \Gamma_2'$ and $\Delta_1 = \Delta_1' + \Delta_2'$, which by inversion must therefore be of the form:

$$\frac{\dfrac{\ldots \qquad \dfrac{}{c : \mathbf{Chan}\,(\mathbf{Send}\, A'\, P) \vdash c : \mathbf{Chan}\,(\mathbf{Send}\, A'\, P)}\ \textsc{var}}{\emptyset, c : \mathbf{Chan}\,(\mathbf{Send}\, A'\, P) \vdash \mathsf{send}\, c : A' \multimap \mathbf{Chan}\, P}\ \multimap_e \qquad \emptyset \vdash v : A'}{\emptyset, c : \mathbf{Chan}\,(\mathbf{Send}\, A'\, P) \vdash \mathsf{send}\, c\, v : \mathbf{Chan}\, P}\ \multimap_e$$

i.e., where $A'' = \mathbf{Chan}\, P$ and $\Gamma_1' = \cdot$ and $\Delta_1' = c : \mathbf{Chan}\,(\mathbf{Send}\, A'\, P)$.
Therefore, for the goal of this theorem we construct the typing:

$$\frac{}{\emptyset, c : \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\ \textsc{var}$$

By Lemma 7 then $\Gamma_2', \Delta_2', c : \mathbf{Chan}\, P \vdash Ctxt\{c\} : B$ which is equal to $\Gamma_1, \Delta_2', c : \mathbf{Chan}\, P \vdash Ctxt\{c\} : B$.
We then build the process configuration type:

$$\dfrac{\dfrac{\Gamma_2', \Delta_2', c : \mathbf{Chan}\, P \vdash Ctxt\{c\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_2' + \Gamma_2, \Delta_2' + \Delta_2, c : \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{c\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R}}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c : \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{c\}^{\mathcal{R}} : A} \equiv \tag{D.3}$$

where the equality step follows by $\Delta_1 = \Delta_1' + \Delta_2' = \Delta_2'$ and $\Gamma_1 = \Gamma_1' + \Gamma_2' = \Gamma_2'$.

2. Typing $[\Gamma_1', \Delta_1'] \vdash \mathsf{send}\, c\, v : A''$ and $\Gamma_2', y : [A'']_r, \Delta_2' \vdash Ctxt\{y\} : B$ with $\Gamma_1 = r * \Gamma_1' + \Gamma_2'$ and $\Delta_1 = \Delta_1' + \Delta_2'$, which by inversion must therefore be of the form:

$$\dfrac{\dfrac{\dfrac{}{c :^* \mathbf{Chan}\,(\mathbf{Send}\, A'\, P) \vdash c : \mathbf{Chan}\,(\mathbf{Send}\, A'\, P)}\ \textsc{varChanS}}{c :^* \mathbf{Chan}\,(\mathbf{Send}\, A'\, P) \vdash \mathsf{send}\, c : A' \multimap \mathbf{Chan}\, P}\ \multimap_e \qquad \emptyset \vdash v : A'}{c :^* \mathbf{Chan}\,(\mathbf{Send}\, A'\, P) \vdash \mathsf{send}\, c\, v : \mathbf{Chan}\, P}\ \multimap_e$$

i.e., where $A'' = \mathbf{Chan}\, P$ and $\Gamma_1' = \cdot$ and $\Delta_1' = c :^* \mathbf{Chan}\,(\mathbf{Send}\, A'\, P)$.
Therefore, for the goal of this theorem we construct the typing:

$$\dfrac{}{c :^* \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\ \textsc{varChanS}$$

By Lemma 8 then $\Gamma_2', \Delta_2', c :^* \mathbf{Chan}\, P \vdash Ctxt\{c\} : B$ which is equal to $\Gamma_1, \Delta_2', c :^* \mathbf{Chan}\, P \vdash Ctxt\{c\} : B$ (since $\Gamma_1 = r * \Gamma_1' + \Gamma_2' = r * \emptyset + \Gamma_2' = \Gamma_2'$).
We then build the process configuration type:

$$\dfrac{\dfrac{\Gamma_2', \Delta_2', c :^* \mathbf{Chan}\, P \vdash Ctxt\{c\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_2' + \Gamma_2, \Delta_2' + \Delta_2, c :^* \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{c\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R}}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c :^* \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{c\}^{\mathcal{R}} : A} \equiv \tag{D.4}$$

where the equality step follows by $\Delta_1 = \Delta_1' + \Delta_2' = \Delta_2'$ and $\Gamma_1 = r * \Gamma_1' + \Gamma_2' = \Gamma_2'$.

– (recv)

$$\dfrac{}{(\mathcal{P} \mid Ctxt\{\mathsf{receive}\, c\};\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{(v, c)\};\ \mathcal{C}, c = \vec{v})}\ \textsc{recv}$$

with process configuration typing:

$$\dfrac{\Gamma_1, \Delta_1, c :^r \mathbf{Chan}\,(\mathbf{Recv}\, A\, P) \vdash Ctxt\{\mathsf{receive}\, c\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c :^r \mathbf{Chan}\,(\mathbf{Recv}\, A\, P) \vdash \mathcal{P} \mid Ctxt\{\mathsf{receive}\, c\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R} \tag{D.5}$$

with channel configuration typing:

$$\dfrac{\Delta_3 \vdash v : A \qquad \Delta_3, \Delta_4, c :^r \mathbf{Chan}\, P \vdash \mathcal{C}, c = \vec{v}}{\Delta_3, \Delta_4, c :^r \mathbf{Chan}\,(\mathbf{Recv}\, A\, P) \vdash \mathcal{C}, c = v : \vec{v}}\ \textsc{recv} \tag{D.6}$$

where $(\Delta_3, \Delta_4) = \Delta_1 + \Delta_2$. (note that EMPTY cannot be used in the root of this derivation as the reduction constraints the channel to start non-empty).
We then construct the consequent channel configuration typing given by the premise of (D.6):

$$\Delta_1 + \Delta_2, c :^r \mathbf{Chan}\, P \vdash \mathcal{C}, c = \vec{v}$$

From typing $\Gamma_1, \Delta_1 \vdash Ctxt\{\mathsf{receive}\, c\} : B$, and by Lemma 6 then there are two possible cases:

1. Typing $\Gamma_1', \Delta_1' \vdash \mathsf{receive}\, c : A''$ and $\Gamma_2', y : A'', \Delta_2' \vdash Ctxt\{y\} : B$ with $\Gamma_1 = \Gamma_1' + \Gamma_2'$ and $\Delta_1 = \Delta_1' + \Delta_2'$, which by inversion must therefore be of the form:

$$\dfrac{\dfrac{}{\emptyset, c : \mathbf{Chan}\,(\mathbf{Recv}\, A\, P) \vdash c : \mathbf{Chan}\,(\mathbf{Recv}\, A\, P)}\ \textsc{var}}{\emptyset, c : \mathbf{Chan}\,(\mathbf{Recv}\, A\, P) \vdash \mathsf{receive}\, c : (A \otimes \mathbf{Chan}\, P)}\ \multimap_e$$

where $A'' = (A \otimes \mathbf{Chan}\, P)$ and $\Gamma_1' = \cdot$ and $\Delta_1' = c : \mathbf{Chan}\,(\mathbf{Recv}\, A\, P)$.

Therefore, for the goal of this theorem we construct the typing:

$$\dfrac{\dfrac{}{c : \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\ \textsc{var} \qquad \emptyset \vdash v : A' \text{ (by (D.6))}}{c : \mathbf{Chan}\, P \vdash (v, c) : (A \otimes \mathbf{Chan}\, P)}\ \otimes_i$$

By Lemma 7 then $\Gamma'_2, \Delta'_2, c : \mathbf{Chan}\, P \vdash Ctxt\{(v, c)\} : B$ which is equal to $\Gamma_1, \Delta'_2, c : \mathbf{Chan}\, P \vdash Ctxt\{(v, c)\} : B$
We then construct the preserved process configuration typing:

$$\dfrac{\dfrac{\Gamma'_2, \Delta'_2, c : \mathbf{Chan}\, P \vdash Ctxt\{(v, c)\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma'_2 + \Gamma_2, \Delta'_2 + \Delta_2, c : \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{(v, c)\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R}}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c : \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{(v, c)\}^{\mathcal{R}} : A}\ \equiv \tag{D.7}$$

where the equality step follows by $\Delta_1 = \Delta'_1 + \Delta'_2 = \Delta'_2$ and $\Gamma_1 = \Gamma'_1 + \Gamma'_2 = \Gamma'_2$.

2. Typing $[\Gamma'_1, \Delta'_1] \vdash \mathsf{receive}\, c : A''$ and $\Gamma'_2, y : [A'']_r, \Delta'_2 \vdash Ctxt\{y\} : B$ with $\Gamma_1 = r * \Gamma'_1 + \Gamma'_2$ and $\Delta_1 = \Delta'_1 + \Delta'_2$, which by inversion must therefore be of the form:

$$\dfrac{\dfrac{}{c :^* \mathbf{Chan}\, (\mathbf{Recv}\, A\, P) \vdash c : \mathbf{Chan}\, (\mathbf{Recv}\, A\, P)}\ \textsc{varChanS}}{c :^* \mathbf{Chan}\, (\mathbf{Recv}\, A\, P) \vdash \mathsf{receive}\, c : (A \otimes \mathbf{Chan}\, P)}\ \multimap_e$$

where $A'' = (A \otimes \mathbf{Chan}\, P)$ and $\Gamma'_1 = \cdot$ and $\Delta'_1 = c :^* \mathbf{Chan}\, (\mathbf{Recv}\, A\, P)$.
Therefore, for the goal of this theorem we construct the typing:

$$\dfrac{\dfrac{}{c :^* \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\ \textsc{varChanS} \qquad \emptyset \vdash v : A' \text{ (by (D.6))}}{c :^* \mathbf{Chan}\, P \vdash (v, c) : (A \otimes \mathbf{Chan}\, P)}\ \otimes_i$$

By Lemma 8 then $\Gamma'_2, \Delta'_2, c :^* \mathbf{Chan}\, P \vdash Ctxt\{(v, c)\} : B$ which is equal to $\Gamma_1, \Delta'_2, c :^* \mathbf{Chan}\, P \vdash Ctxt\{(v, c)\} : B$ (since $\Gamma_1 = r * \Gamma'_1 + \Gamma'_2 = r * \emptyset + \Gamma'_2 = \Gamma'_2$).
We then construct the preserved process configuration typing:

$$\dfrac{\dfrac{\Gamma'_2, \Delta'_2, c :^* \mathbf{Chan}\, P \vdash Ctxt\{(v, c)\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma'_2 + \Gamma_2, \Delta'_2 + \Delta_2, c :^* \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{(v, c)\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R}}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c :^* \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{(v, c)\}^{\mathcal{R}} : A}\ \equiv \tag{D.8}$$

where the equality step follows by $\Delta_1 = \Delta'_1 + \Delta'_2 = \Delta'_2$ and $\Gamma_1 = r * \Gamma'_1 + \Gamma'_2 = \Gamma'_2$.

– (close)

$$\dfrac{}{(\mathcal{P} \mid Ctxt\{\mathsf{close}\, c\};\ \mathcal{C}, c^1 = \langle\rangle) \rightsquigarrow (\mathcal{P} \mid Ctxt\{\mathbf{unit}\};\ \mathcal{C})}\ \textsc{close}$$

with process configuration typing:

$$\dfrac{\dfrac{\Gamma_1, \Delta_1, c : \mathbf{Chan}\, \mathbf{End} \vdash Ctxt\{\mathsf{close}\, c\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c : \mathbf{Chan}\, \mathbf{End} \vdash \mathcal{P} \mid Ctxt\{\mathsf{close}\, c\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R}}{} \tag{D.9}$$

(i.e., $c \notin \mathsf{dom}(\Delta_2)$) with only one potential channel configuration typing:

$$\dfrac{\dfrac{\Delta_1 + \Delta_2 \vdash \mathcal{C}}{\Delta_1 + \Delta_2, c : \mathbf{Chan}\, \mathbf{End} \vdash \mathcal{C}, c^1 = \langle\rangle}\ \textsc{empty}}{} \tag{D.10}$$

The goal channel configuration $\Delta_1 + \Delta_2 \vdash \mathcal{C}$ is then given by the premise of (D.10).
From typing $\Gamma_1, \Delta_1 \vdash Ctxt\{\mathsf{close}\, c\} : B$, and by Lemma 6, and since we know here that c is a linear channel, then the typing factorises to the typing $\Gamma'_1, \Delta'_1 \vdash \mathsf{close}\, c : A''$ and $\Gamma'_2, y : A'', \Delta'_2 \vdash Ctxt\{y\} : B$ with $\Gamma_1 = \Gamma'_1 + \Gamma'_2$ and $\Delta_1 = \Delta'_1 + \Delta'_2$, which by inversion must therefore be of the form:

$$\dfrac{\dfrac{}{c : \mathbf{Chan}\, \mathbf{End} \vdash c : \mathbf{Chan}\, \mathbf{End}}}{c : \mathbf{Chan}\, \mathbf{End} \vdash \mathsf{close}\, c : 1}\ \multimap_e$$

where $A'' = 1$ and $\Gamma'_1 = \cdot$ and $\Delta'_1 = c : \mathbf{Chan}\, \mathbf{End}$.

Therefore, for the goal of this theorem we construct the typing:

$$\frac{}{\emptyset \vdash \mathbf{unit} : 1}\ \text{UNIT}_i$$

By Lemma 7 then $\Gamma_2', \Delta_2' \vdash Ctxt\{\mathbf{unit}\} : B$.
We then construct the preserved process configuration typing:

$$\frac{\dfrac{\Gamma_2', \Delta_2' \vdash Ctxt\{\mathbf{unit}\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_2' + \Gamma_2, \Delta_2' + \Delta_2 \vdash \mathcal{P} \mid Ctxt\{\mathbf{unit}\}^{\mathcal{R}} : A}\ \text{CONS}\mathcal{R}}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash \mathcal{P} \mid Ctxt\{\mathbf{unit}\}^{\mathcal{R}} : A}\ \equiv \tag{D.11}$$

where the equality step follows by $\Delta_1 = \Delta_1' + \Delta_2' = \Delta_2'$ and $\Gamma_1 = \Gamma_1' + \Gamma_2' = \Gamma_2'$.

– (sharedClose)

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{close}\, c\};\ \mathcal{C}, c^* = \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{\mathbf{unit}\};\ \mathcal{C}, c^* = \vec{v})}\ \text{SHAREDCLOSE}$$

with process configuration typing:

$$\frac{\Gamma_1, \Delta_1, c :^* \mathbf{Chan\ End} \vdash Ctxt\{\mathsf{close}\, c\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c :^* \mathbf{Chan\ End} \vdash \mathcal{P} \mid Ctxt\{\mathsf{close}\, c\}^{\mathcal{R}} : A}\ \text{CONS}\mathcal{R} \tag{D.12}$$

and channel configuration typing $\Delta_1 + \Delta_2, c :^* \mathbf{Chan\ End} \vdash \mathcal{C}, c^* = \vec{v}$.
Since the reduction semantics preserves the channel configuration, this channel configuration typing then also provides the goal configuration typing as-is.
From typing $\Gamma_1, \Delta_1 \vdash Ctxt\{\mathsf{close}\, c\} : B$, and by Lemma 6, and since we know here that c is a non-linear channel, then the typing factorises to the typing $[\Gamma_1', \Delta_1'] \vdash \mathsf{close}\, c : A''$ and $\Gamma_2', y : [A'']_r, \Delta_2' \vdash Ctxt\{y\} : B$ with $\Gamma_1 = r * \Gamma_1' + \Gamma_2'$ and $\Delta_1 = \Delta_1' + \Delta_2'$, which by inversion must therefore be of the form:

$$\frac{c :^* \mathbf{Chan\ End} \vdash c : \mathbf{Chan\ End}}{c :^* \mathbf{Chan\ End} \vdash \mathsf{close}\, c : 1}\ \multimap_e$$

where $A'' = 1$ and $\Gamma_1' = \cdot$ and $\Delta_1' = c :^* \mathbf{Chan\ End}$.
Therefore, for the goal of this theorem we construct the typing:

$$\frac{\dfrac{}{\emptyset \vdash \mathbf{unit} : 1}\ \text{UNIT}_i}{c :^* \mathbf{Chan\ End} \vdash \mathbf{unit} : 1}\ \text{WEAKCHANS}$$

By Lemma 8 then $\Gamma_2', \Delta_2', c :^* \mathbf{Chan\ End} \vdash Ctxt\{\mathbf{unit}\} : B$.
We then construct the preserved process configuration typing:

$$\frac{\dfrac{\Gamma_2', \Delta_2', c :^* \mathbf{Chan\ End} \vdash Ctxt\{\mathbf{unit}\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_2' + \Gamma_2, \Delta_2' + \Delta_2, c :^* \mathbf{Chan\ End} \vdash \mathcal{P} \mid Ctxt\{\mathbf{unit}\}^{\mathcal{R}} : A}\ \text{CONS}\mathcal{R}}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c :^* \mathbf{Chan\ End} \vdash \mathcal{P} \mid Ctxt\{\mathbf{unit}\}^{\mathcal{R}} : A}\ \equiv \tag{D.13}$$

where the equality step follows by $\Delta_1 = \Delta_1' + \Delta_2' = \Delta_2'$ and $\Gamma_1 = r * \Gamma_1' + \Gamma_2' = \Gamma_2'$.

– (fork)

$$\frac{\#c}{(\mathcal{P} \mid Ctxt\{\mathsf{forkLinear}\,(\lambda x.t)\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^\circ \mid Ctxt\{\overline{c}\};\ \mathcal{C}, \overline{c} = \langle\rangle, c = \langle\rangle)}\ \text{FORKLINEAR}$$

with process configuration typing:

$$\frac{\Gamma_1, \Delta_1 \vdash Ctxt\{\mathsf{forkLinear}\,(\lambda x.t)\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash \mathcal{P} \mid Ctxt\{\mathsf{forkLinear}\,(\lambda x.t)\}^{\mathcal{R}} : A}\ \text{CONS}\mathcal{R} \tag{D.14}$$

with channel configuration typing:

$$\Delta_1 + \Delta_2 \vdash \mathcal{C} \tag{D.15}$$

From typing $\Gamma_1, \Delta_1 \vdash Ctxt\{\mathsf{forkLinear}\,(\lambda x.t)\} : B$, and by Lemma 6 there are two possibilities:

1. (second disjunct) (with $Ctxt\{\mathsf{forkLinear}\,(\lambda x.t)\}$ factorising into a graded binding) Gives $\neg\mathsf{resourceAllocator}(\mathsf{forkLinear}\,(\lambda x.t))$ which is a contradiction, thus we can conclude trivial by ex falso.

2. (first disjunct) Typing $\Gamma_1', \Delta_1' \vdash \mathsf{forkLinear}\,(\lambda x.t) : A$, which by inversion must therefore be of the form:

$$\dfrac{\dfrac{\Gamma_1', x : \mathbf{Chan}\,P, \Delta_1' \vdash t : 1}{\Gamma_1', \Delta_1' \vdash (\lambda x.t) : \mathbf{Chan}\,P \multimap 1}\ \multimap_i}{\Gamma_1', \Delta_1' \vdash \mathsf{forkLinear}\,(\lambda x.t) : \mathbf{Chan}\,(\overline{P})}\ \multimap_e \tag{D.16}$$

where $A = \mathbf{Chan}\,(\overline{P})$ and $\Gamma_1'', x : \mathbf{Chan}\,(\overline{P}), \Delta_1'' \vdash Ctxt\{x\} : B$ where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and $\Delta_1 = \Delta_1' + \Delta_1''$.
Therefore, for the goal of this theorem we construct the typings:

$$\dfrac{}{\emptyset, \overline{c} : \mathbf{Chan}\,(\overline{P}) \vdash \overline{c} : \mathbf{Chan}\,(\overline{P})}\ \textsc{var}$$

and

$$\dfrac{}{\emptyset, c : \mathbf{Chan}\,P \vdash c : \mathbf{Chan}\,P}\ \textsc{var} \tag{D.17}$$

By Lemma 7 then $\Gamma_1'', \overline{c} : \mathbf{Chan}\,(\overline{P}) \vdash Ctxt\{\overline{c}\} : B$.
We construct the goal channel configuration typing as:

$$\dfrac{\dfrac{\Delta_1 \vdash \mathcal{C}}{\Delta_1 + \Delta_2, c : \mathbf{Chan}\,P \vdash \mathcal{C}, c = \langle\rangle}\ \textsc{empty}}{\Delta_1 + \Delta_2, c : \mathbf{Chan}\,P, \overline{c} : \mathbf{Chan}\,(\overline{P}) \vdash \mathcal{C}, c = \langle\rangle, \overline{c} = \langle\rangle}\ \textsc{empty}$$

with process configuration typing then given by:

$$\dfrac{\dfrac{\dfrac{\Gamma_1'', \Delta_1'', \overline{c} : \mathbf{Chan}\,(\overline{P}) \vdash Ctxt\{\overline{c}\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1'' + \Gamma_2, \Delta_1'' + \Delta_2, \overline{c} : \mathbf{Chan}\,(\overline{P}) \vdash \mathcal{P} \mid Ctxt\{\overline{c}\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R} \qquad \dfrac{(\text{D.16})\ (\text{D.17})}{\Gamma_1', \Delta_1', c : \mathbf{Chan}\,P \vdash t[c/x] : 1}\ \textsc{subst}}{\Gamma_1'' + \Gamma_2 + \Gamma_1', \Delta_1' + \Delta_2 + \Delta_1'', c : \mathbf{Chan}\,P, \overline{c} : \mathbf{Chan}\,(\overline{P}) \vdash \mathcal{P} \mid t[c/x]^\circ \mid Ctxt\{\overline{c}\}^{\mathcal{R}} : A}\ \textsc{sub}}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c : \mathbf{Chan}\,P, \overline{c} : \mathbf{Chan}\,(\overline{P}) \vdash \mathcal{P} \mid t[c/x]^\circ \mid Ctxt\{\overline{c}\}^{\mathcal{R}} : A}\ \equiv$$

where disjointness conditions in the above for c and $\overline{c}$ are satisfied by the fact that c is fresh as per the semantics, and disjointness of the constituent contexts comes from their original disjointness being defined (e.g., $\Delta_1 = \Delta_1' + \Delta_1''$).

– (forkNonLinear)

$$\dfrac{\#c}{(\mathcal{P} \mid Ctxt\{\mathsf{forkNonLinear}\,(\lambda x.t)\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{[\overline{c}]\} \mid t[[c]/x]^\circ;\ \mathcal{C}, \overline{c}^* = \langle\rangle, c^* = \langle\rangle)}\ \textsc{forkNonLinear}$$

with process configuration typing:

$$\dfrac{\Gamma_1, \Delta_1 \vdash Ctxt\{\mathsf{forkNonLinear}\,(\lambda x.t)\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash \mathcal{P} \mid Ctxt\{\mathsf{forkNonLinear}\,(\lambda x.t)\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R} \tag{D.18}$$

with channel configuration typing:

$$\Delta_1 + \Delta_2 \vdash \mathcal{C} \tag{D.19}$$

From typing $\Gamma_1, \Delta_1 \vdash Ctxt\{\mathsf{forkNonLinear}\,(\lambda x.t)\} : B$, and by Lemma 6 there are two possibilities:
1. (second disjunct) (with $Ctxt\{\mathsf{forkNonLinear}\,(\lambda x.t)\}$ factorising into a graded binding) Gives $\neg\mathsf{resourceAllocator}(\mathsf{forkNonLinear}\,(\lambda x.t))$ which is a contradiction, thus we can conclude trivial by ex falso.
2. (first disjunct) There is a typing $\Gamma_1', \Delta_1' \vdash \mathsf{forkNonLinear}\,(\lambda x.t) : A$, which by inversion must therefore be of the form:

$$\dfrac{\dfrac{\Gamma_1', x : \Box_r(\mathbf{Chan}\,P), \Delta_1' \vdash t : 1}{\Gamma_1', \Delta_1' \vdash (\lambda x.t) : \Box_r(\mathbf{Chan}\,P) \multimap 1}\ \multimap_i}{\Gamma_1', \Delta_1' \vdash \mathsf{forkNonLinear}\,(\lambda x.t) : \Box_r(\mathbf{Chan}\,(\overline{P}))}\ \multimap_e \tag{D.20}$$

where $A = \Box_r(\mathbf{Chan}\,(\overline{P}))$ and $\Gamma_1'', x : \Box_r(\mathbf{Chan}\,(\overline{P})), \Delta_1'' \vdash Ctxt\{x\} : B$ and where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and $\Delta_1 = \Delta_1' + \Delta_1''$.
For the goal of this theorem we construct the typings:

$$\dfrac{\dfrac{}{\emptyset, \overline{c} :^* \mathbf{Chan}\,(\overline{P}) \vdash \overline{c} : \mathbf{Chan}\,(\overline{P})}\ \textsc{var}}{\emptyset, \overline{c} :^* \mathbf{Chan}\,(\overline{P}) \vdash [\overline{c}] : \Box_r(\mathbf{Chan}\,(\overline{P}))}\ \textsc{pr} \tag{D.21}$$

and

$$\frac{\dfrac{}{\emptyset, c :^* \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\,\text{VAR}}{\emptyset, c :^* \mathbf{Chan}\, P \vdash [c] : \Box_r(\mathbf{Chan}\, P)}\,\text{PR} \tag{D.22}$$

By Lemma 7 then $\Gamma_1'', \Delta_1'', \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash Ctxt\{[\overline{c}]\} : B$.
We then construct the consequent channel configuration typing:

$$\frac{\dfrac{\dfrac{}{\Delta_1 + \Delta_2 \vdash \mathcal{C}}(\text{D.19})}{\Delta_1 + \Delta_2, c :^* \mathbf{Chan}\, P \vdash \mathcal{C}, c^* = \langle\rangle}\,\text{EMPTY}}{\Delta_1 + \Delta_2, c :^* \mathbf{Chan}\, P, \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash \mathcal{C}, c^* = \langle\rangle, \overline{c^*} = \langle\rangle}\,\text{EMPTY}$$

and process configuration:

$$\frac{\dfrac{\Gamma_1'', \Delta_1'', \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash Ctxt\{[\overline{c}]\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1'' + \Gamma_2, \Delta_1'' + \Delta_2, \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash \mathcal{P} \mid Ctxt\{[\overline{c}]\}^{\mathcal{R}} : A}\,\text{CONS}\mathcal{R} \qquad \dfrac{(\text{D.20})\ (\text{D.22})}{\Gamma_1', \Delta_1', c :^* \mathbf{Chan}\, P \vdash t[[c]/x] : 1}\,\text{GRADED-SUBST}}{\Gamma_1' + \Gamma_1'' + \Gamma_2, \Delta_1' + \Delta_1'' + \Delta_2, c :^* \mathbf{Chan}\, P, \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash \mathcal{P} \mid Ctxt\{[\overline{c}]\}^{\mathcal{R}} \mid t[[c]/x]^\circ : A}\,\text{SUB}}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c :^* \mathbf{Chan}\, P, \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash \mathcal{P} \mid Ctxt\{[\overline{c}]\}^{\mathcal{R}} \mid t[[c]/x]^\circ : A} \equiv$$

where disjointness conditions in the above for c and $\overline{c}$ are satisfied by the fact that c is fresh as per the semantics, and disjointness of the constituent contexts comes from their original disjointness being defined (e.g., $\Delta_1 = \Delta_1' + \Delta_1''$.

– (forkReplicate)

$$\frac{\#c_1 \ldots \#c_n}{(\mathcal{P} \mid Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{\langle[\overline{c_1}], \ldots, [\overline{c_n}]\rangle\} \mid \mathsf{Wch}\,(c_1 \mapsto \lambda x.t) \mid \ldots \mid \mathsf{Wch}\,(c_n \mapsto \lambda x.t);\ \mathcal{C}, c_1 = \langle\rangle, \ldots, c_n = \langle\rangle, \overline{c_1} = \langle\rangle, \ldots, \overline{c_n} = \langle\rangle)}\,\text{FORKREPLICATE}$$

with process configuration typing:

$$\frac{\Gamma_1, \Delta_1 \vdash Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash \mathcal{P} \mid Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\}^{\mathcal{R}} : A}\,\text{CONS}\mathcal{R} \tag{D.23}$$

with channel configuration typing:

$$\Delta_1 + \Delta_2 \vdash \mathcal{C} \tag{D.24}$$

From typing $\Gamma_1, \Delta_1 \vdash Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\} : B$, and by Lemma 6 there are two possibilities:

1. (second disjunct) (with $Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\}$ factorising into a graded binding) Gives $\neg\mathsf{resourceAllocator}(\mathsf{forkReplicate}\,[\lambda x.t]\,n)$ which is a contradiction, thus we can conclude trivial by ex falso.
2. (first disjunct) then there is a typing $\Gamma_1'', \Delta_1'' \vdash \mathsf{forkReplicate}\,[\lambda x.t]\,n : A$, which by inversion must therefore be of the form:

$$\frac{\dfrac{\dfrac{\dfrac{\Gamma_1'', x : \mathbf{Chan}\, P, \Delta_1'' \vdash t : 1}{\Gamma_1'', \Delta_1'' \vdash \lambda x.t : \mathbf{Chan}\, P \multimap 1}\,\multimap_i}{(0..n) * \Gamma_1'', \Delta_1'' \vdash [\lambda x.t] : \Box_{0..n}(\mathbf{Chan}\, P \multimap 1)}\,\text{PR}}{(0..n) * \Gamma_1'', \Delta_1'' \vdash \mathsf{forkReplicate}\,[\lambda x.t] : \mathbf{N}\, n \multimap \mathbf{Vec}\, n\, (\Box_{0..1}(\mathbf{Chan}\, (\overline{P})))}\,\multimap_e}{(0..n) * \Gamma_1'', \Delta_1'' \vdash \mathsf{forkReplicate}\,[\lambda x.t]\,n : \mathbf{Vec}\, n\, (\Box_{0..1}(\mathbf{Chan}\, (\overline{P})))}\,\multimap_e \tag{D.25}$$

where $A = \mathbf{Vec}\, n\, (\Box_{0..1}(\mathbf{Chan}\, (\overline{P})))$ and $\Gamma_1', x : \mathbf{Vec}\, n\, (\Box_{0..1}(\mathbf{Chan}\, (\overline{P}))), \Delta_1' \vdash Ctxt\{x\} : B$ and where $\Delta_1 = \Delta_1' + \Delta_1''$ and where $\Gamma_1 = \Gamma_1' + (0..n) * \Gamma_1''$
Furthermore by the ReceivePrefix(P) constraint of forkReplicate we know that $P = \mathbf{Recv}\, A'\, P'$.
For the goal of this theorem we construct the typings:

$$\frac{\ldots}{\overline{c_1} :^* \mathbf{Chan}\, (\mathbf{Send}\, A'\, P'), \ldots, \overline{c_n} :^* \mathbf{Chan}\, (\mathbf{Send}\, A'\, P') \vdash \langle[\overline{c_1}], \ldots, [\overline{c_n}]\rangle : \mathbf{Vec}\, n\, (\Box_{0..1}(\mathbf{Chan}\, (\mathbf{Send}\, A'\, P')))}\,\text{CONS}^n + \text{NIL}$$

By Lemma 7 then:
$\Gamma_1', \Delta_1', \overline{c_1} :^* \mathbf{Chan}\, (\mathbf{Send}\, A'\, P'), \ldots, \overline{c_n} :^* \mathbf{Chan}\, (\mathbf{Send}\, A'\, P') \vdash Ctxt\{\langle[\overline{c_1}], \ldots, [\overline{c_n}]\rangle\} : B$.
The consequent channel configuration has type derivation (and as a short hand we write $c_i : \mathbf{Chan}\, P_1$ for $c_1 :^* \mathbf{Chan}\, P_1, \ldots, c_n :^* \mathbf{Chan}\, P_n$):

$$\frac{\Delta_1 + \Delta_2 \vdash \mathcal{C}}{\Delta_1 + \Delta_2, c_i : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P'), \overline{c_i} :^* \mathbf{Chan}\,(\mathbf{Send}\,A'\,P') \vdash \mathcal{C}, {c_1}^* = \langle\rangle, .., {c_n}^* = \langle\rangle, \overline{c_1}^* = \langle\rangle, .., \overline{c_n}^* = \langle\rangle}\ \text{EMPTY}^*$$

We then construct the process configuration typing where $\Delta' = \Delta, c_i :^* \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P'), \overline{c_i} :^* \mathbf{Chan}\,(\mathbf{Send}\,A'\,\overline{P'})$

$$\frac{\dfrac{\dfrac{\Gamma'_1, \Delta'_1, \overline{c_i} :^* \mathbf{Chan}\,(\mathbf{Send}\,A'\,\overline{P'}) \vdash Ctxt\{\langle[\overline{c_1}], .., [\overline{c_n}]\rangle\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma'_1 + \Gamma_2, \Delta'_1 + \Delta_2, \overline{c_i} :^* \mathbf{Chan}\,(\mathbf{Send}\,A'\,\overline{P'}) \vdash \mathcal{P} \mid Ctxt\{\langle[\overline{c_1}], .., [\overline{c_n}]\rangle\}^{\mathcal{R}} : A}\ \text{CONS}\mathcal{R} \qquad \dfrac{}{\Gamma''_1, x : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P'), \Delta''_1 \vdash t : 1}\ (\text{D.25})}{\Gamma'_1 + (0..1 * \Gamma''_1) + \Gamma_2, \Delta'_1 + \Delta''_1 + \Delta_2, \Delta' \vdash \mathcal{P} \mid Ctxt\{\langle[\overline{c_1}], .., [\overline{c_n}]\rangle\}^{\mathcal{R}} \mid \mathsf{Wch}\,(c_1 \mapsto \lambda x.t) : A}\ \text{WATCH}}{\Gamma'_1 + (0..n * \Gamma''_1) + \Gamma_2, \Delta'_1 + \Delta''_1 + \Delta_2, \Delta' \vdash \mathcal{P} \mid Ctxt\{\langle[\overline{c_1}], .., [\overline{c_n}]\rangle\}^{\mathcal{R}} \mid \mathsf{Wch}\,(c_1 \mapsto \lambda x.t) \mid .. \mid \mathsf{Wch}\,(c_n \mapsto \lambda x.t) : A}\ \text{WATCH}^*}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, \Delta' \vdash \mathcal{P} \mid Ctxt\{\langle[\overline{c_1}], .., [\overline{c_n}]\rangle\}^{\mathcal{R}} \mid \mathsf{Wch}\,(c_1 \mapsto \lambda x.t) \mid .. \mid \mathsf{Wch}\,(c_n \mapsto \lambda x.t) : A} \equiv$$

since $0..1 + \ldots + 0..1$ n-times is equal to $0..n$, satisfying the typing goal, and where disjointness conditions in the above for c_i and $\overline{c_i}$ are satisfied by the fact that c_i are all fresh as per the semantics, and disjointness of the constituent contexts comes from their original disjointness being defined (e.g., $\Delta_1 = \Delta'_1 + \Delta''_1$.

– (watch)

$$\frac{}{(\mathcal{P} \mid \mathsf{Wch}\,(c \mapsto \lambda x.t);\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^\circ;\ \mathcal{C}, c = v : \vec{v})}\ \text{WATCHER}$$

with process configuration typing:

$$\frac{\Gamma_1, x : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P), \Delta_1 \vdash t : 1 \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{0..1 * \Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \vdash \mathcal{P} \mid \mathsf{Wch}\,(c \mapsto \lambda x.t) : A}\ \text{WATCH} \tag{D.26}$$

and channel configuration typing:

$$\frac{\Delta_1 + \Delta_2 \vdash \mathcal{C} \qquad \Delta_1 + \Delta_2 \vdash v : A'}{\Delta_1 + \Delta_2, c : (\mathbf{Recv}\,A'\,P) \vdash \mathcal{C}, c = v : \vec{v}} \tag{D.27}$$

For the goal of this theorem we construct the typing:

$$\frac{\dfrac{\Gamma_1, x : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P), \Delta_1 \vdash t : 1 \quad (\text{D.26}) \qquad \dfrac{}{c : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \vdash c : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P)}\ \text{VAR}}{1..1 * \Gamma_1, \Delta_1, c : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \vdash t[c/x] : 1}\ \text{SUBST}}{0..1 * \Gamma_1, \Delta_1, c : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \vdash t[c/x] : 1}\ \text{APPROX} \tag{D.28}$$

From this we can construct the channel configuration typing as just (D.27) and the process config typing:

$$\frac{\Gamma_2, \Delta_2 \vdash \mathcal{P} : A \qquad 0..1 * \Gamma_1, \Delta_1, c : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \vdash t[c/x] : 1\ (\text{D.28})}{0..1 * \Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, c : \mathbf{Chan}\,(\mathbf{Recv}\,A'\,P) \vdash \mathcal{P} \mid t[c/x]^\circ : A}\ \text{WATCH}$$

and the premise channel configuration provides the consequent channel configuration.

– (forkMulticast)

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{forkMulticast}\,(\lambda x.t)\,n\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{\langle\overline{c_1}, .., \overline{c_n}\rangle\} \mid t[c/x]^\circ \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n);\ \mathcal{C}, c = \langle\rangle, \overline{c} = \langle\rangle, c_1 = \langle\rangle, .., c_n = \langle\rangle, \overline{c_1} = \langle\rangle, .., \overline{c_n} = \langle\rangle)}\ \text{FORKMULTICAST}$$

with process configuration typing: process configuration typing:

$$\frac{\Gamma_1, \Delta_1 \vdash Ctxt\{\mathsf{forkMulticast}\,(\lambda x.t)\,n\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2 \vdash \mathcal{P} \mid Ctxt\{\mathsf{forkMulticast}\,(\lambda x.t)\,n\}^{\mathcal{R}} : A}\ \text{CONS}\mathcal{R} \tag{D.29}$$

with channel configuration typing:

$$\Delta_1 + \Delta_2 \vdash \mathcal{C} \tag{D.30}$$

From typing $\Gamma_1, \Delta_1 \vdash Ctxt\{\mathsf{forkMulticast}\,(\lambda x.t)\,n\} : B$, and by Lemma 6 there are two possibilities:

1. (second disjunct) (with $Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\}$ factorising into a graded binding) Gives $\neg\mathsf{resourceAllocator}(\mathsf{forkReplicate}\,[\lambda x.t]\,n)$ which is a contradiction, thus we can conclude trivial by ex falso.
2. (first disjunct) Then there is a typing $\Gamma'_1, \Delta'_1 \vdash \mathsf{forkMulticast}\,(\lambda x.t)\,n : A$, which by inversion must therefore be of the form:

$$\frac{\dfrac{\dfrac{\Gamma'_1, x : \mathbf{Chan}\,(\mathsf{Graded}_n P'), \Delta'_1 \vdash t : 1}{\Gamma'_1, \Delta'_1 \vdash \lambda x.t : \mathbf{Chan}\,(\mathsf{Graded}_n P') \multimap 1}\ \multimap_i}{\Gamma'_1, \Delta'_1 \vdash \mathsf{forkMulticast}\,(\lambda x.t) : \mathbf{N}\,n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P'})}\ \multimap_e}{\Gamma'_1, \Delta'_1 \vdash \mathsf{forkMulticast}\,(\lambda x.t)\,n : \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P'})}\ \multimap_e \tag{D.31}$$

where $A = \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})$ and $\Gamma_1'', x : \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P}), \Delta_1'' \vdash Ctxt\{x\} : B$ and $P = \mathbf{Graded}\, n\, P'$ and where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and where $\Delta_1 = \Delta_1' + \Delta_1''$.
For the goal of this theorem we construct the typing:

$$\frac{\cdots}{\overline{c_1} : \mathbf{Chan}\, \overline{P}, .., \overline{c_n} : \mathbf{Chan}\, \overline{P} \vdash \langle \overline{c_1}, .., \overline{c_n} \rangle : \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})}\ \text{CONS}^n + \text{NIL}$$

By Lemma 7 then $\Gamma_1'', \Delta_1'', \Delta' \vdash Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\} : B$
The consequent channel configuration has type derivation where we write $\Delta' = c_1 : \mathbf{Chan}\, P, .., c_n : \mathbf{Chan}\, P$ and $\overline{\Delta'} = \overline{c_1} : \mathbf{Chan}\, \overline{P}, .., \overline{c_n} : \mathbf{Chan}\, \overline{P}$.

$$\frac{\Delta_1 + \Delta_2 \vdash \mathcal{C}}{\Delta_1 + \Delta_2, c : \mathbf{Chan}\, (\mathsf{Graded}_n P), \overline{c} : \mathbf{Chan}\, \overline{(\mathsf{Graded}_n P)}, \Delta', \overline{\Delta'} \vdash \mathcal{C}, c = \langle\rangle, \overline{c} = \langle\rangle, c_1 = \langle\rangle, .., c_n = \langle\rangle, \overline{c_1} = \langle\rangle, .., \overline{c_n} = \langle\rangle}\ \text{EMPTY}^*$$

We then construct the process configuration typing:

$$\frac{\dfrac{\dfrac{\Gamma_1'', \Delta_1'', \Delta' \vdash Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\} : B \qquad \Gamma_2, \Delta_2 \vdash \mathcal{P} : A}{\Gamma_1'' + \Gamma_2, \Delta_1'' + \Delta_2, \Delta' \vdash \mathcal{P} \mid Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\}^{\mathcal{R}} : A}\ \text{CONS}\mathcal{R} \qquad \dfrac{\Gamma_1', x : \mathbf{Chan}\, (\mathsf{Graded}_r P'), \Delta_1' \vdash t : 1}{\Gamma_1', \Delta_1', c : \mathbf{Chan}\, (\mathsf{Graded}_n P') \vdash t[c/x] : 1}\ \text{SUBST}}{\Gamma_1' + \Gamma_1'' + \Gamma_2, \Delta_1' + \Delta_1'' + \Delta_2, c : \mathbf{Chan}\, (\mathsf{Graded}_n P'), \overline{\Delta'} \vdash \mathcal{P} \mid Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\}^{\mathcal{R}} \mid t[c/x]^\circ : A}\ \text{CONS}\mathcal{R}}{\Gamma_1' + \Gamma_1'' + \Gamma_2, \Delta_1' + \Delta_1'' + \Delta_2, \Delta', \overline{\Delta'}, c : \mathbf{Chan}\, (\mathsf{Graded}_n P'), \overline{c} : \mathbf{Chan}\, \overline{(\mathsf{Graded}_n P')} \vdash \mathcal{P} \mid Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\}^{\mathcal{R}} \mid t[c/x]^\circ \mid \mathsf{Fwd}\, (c \mapsto c_1, .., c_n) : A}\ \text{FWD}}{\Gamma_1 + \Gamma_2, \Delta_1 + \Delta_2, \Delta', \overline{\Delta'}, c : \mathbf{Chan}\, (\mathsf{Graded}_n P'), \overline{c} : \mathbf{Chan}\, \overline{(\mathsf{Graded}_n P')} \vdash \mathcal{P} \mid Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\}^{\mathcal{R}} \mid t[c/x]^\circ \mid \mathsf{Fwd}\, (c \mapsto c_1, .., c_n) : A}\ \equiv$$

satisfying the goal.

– (forward)

$$\frac{}{(\mathcal{P} \mid \mathsf{Fwd}\, (c \mapsto c_1, .., c_n);\ \mathcal{C}, c = [v] : \vec{v}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n) \rightsquigarrow (\mathcal{P} \mid \mathsf{Fwd}\, (c \mapsto c_1, .., c_n);\ \mathcal{C}, c = \vec{v}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v)}\ \text{FORWARDER}$$

with process configuration typing:

$$\frac{\mathsf{Sends}(P) \qquad \Gamma, \Delta \vdash \mathcal{P} : A}{\Gamma, \Delta, c : \mathbf{Chan}\, (\mathsf{Graded}_n \overline{P}), c_i : \mathbf{Chan}\, P \vdash \mathcal{P} \mid \mathsf{Fwd}\, (c \mapsto c_1, .., c_n) : A}\ \text{FWD} \tag{D.32}$$

(specialisation of the FWD rule since $\mathsf{Sends}(\overline{\mathbf{Recv}\, A'\, P}) = \mathsf{Sends}(\overline{P})$) and with channel configuration typing:

$$\frac{\dfrac{\Delta_0 \vdash v : A'}{\Delta_0 \vdash [v] : \Box_n A'}\ \text{PR} \qquad \dfrac{\Delta_0 \vdash v : A' \quad \Delta_0, \Delta_1, c_1 : \mathbf{Chan}\, P, .., c_n : \mathbf{Chan}\, P, c : \mathbf{Chan}\, \overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v, c = \vec{v}}{\Delta_0, \Delta_1, c_1 : \mathbf{Chan}\, (\mathbf{Send}\, A\, P), .., c_n : \mathbf{Chan}\, (\mathbf{Send}\, A\, P), c : \mathbf{Chan}\, \overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n, c = \vec{v}}\ \text{SEND}^n}{\Delta_0, \Delta_1, c_1 : \mathbf{Chan}\, (\mathbf{Send}\, A\, P), .., c_n : \mathbf{Chan}\, (\mathbf{Send}\, A\, P), c : \mathbf{Chan}\, (\mathbf{Recv}\, (\Box_n A')\, \overline{P}) \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n, c = [v] : \vec{v}}\ \text{RECV} \tag{D.33}$$

where $\Delta = \Delta_0, \Delta_1$.
The goal channel configuration typing is then given by the premise of (D.33):

$$\frac{\Delta_0 \vdash v : A' \quad \Delta_1, c_1 : \mathbf{Chan}\, P, .., c_n : \mathbf{Chan}\, P, c : \mathbf{Chan}\, \overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v, c = \vec{v}}{\Delta_0, \Delta_1, c_1 : \mathbf{Chan}\, (\mathbf{Send}\, A\, P), .., c_n : \mathbf{Chan}\, (\mathbf{Send}\, A\, P), c : \mathbf{Chan}\, \overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n, c = \vec{v}}\ \text{SEND}^n$$

From the definition of $\mathsf{Sends}(\overline{\mathbf{Recv}\, A'\, P}) = \mathsf{Sends}(\overline{P})$ and $\mathsf{Sends}(\mathbf{Send}\, A'\, P) = \mathsf{Sends}(P)$ therefore we can define the process configuration typing as:

$$\frac{\mathsf{Sends}((\overline{P})) \qquad \mathsf{Sends}(P) \qquad \Gamma, \Delta \vdash \mathcal{P} : A}{\Gamma, \Delta, \overline{c} : \mathbf{Chan}\, (\overline{P}), c_1 : \mathbf{Chan}\, P, .., c_n : \mathbf{Chan}\, P \vdash \mathcal{P} \mid \mathsf{Fwd}\, (c \mapsto c_1, .., c_n) : A}\ \text{FWD} \qquad \square$$

Appendix E. Related work further details; comparison with GV

The construction in GV of the replicate primitive can also be captured in Granule. Their construction [11, Appendix B] can be written in Granule with $\Box A = A$ `[Many]` (where $\mathcal{R} = \{\mathtt{Zero}, \mathtt{One}, \mathtt{Many}\}$ is a semiring for capturing linear/non-linear behaviour [47]) but only if we both turn *off* the resource allocator restriction on promotion that we introduced in Section 4, which was critical for soundness in our call-by-value semantics, and use call-by-name semantics, both of which are provided by a language extension:

```
1  language CBN -- Call-by-Name semantics, also turns off the restriction
2               -- to non-resource allocating terms in promotion
3
4  -- GV 'replicate'
5  replicate : ∀ {s : Protocol}
6            . (LChan (Send ((LChan (Dual s)) [Many]) End)
7            , (LChan s → ()) [Many]) → LChan End
```

```
8  replicate (x, f) =
9  let [g] = f in send x [forkLinear g]
10
11 -- Dual 'request' primitive
12 request : ∀ {s : Protocol}
13          . LChan (Recv ((LChan s) [Many]) End) → LChan s
14 request s = let (w , s) = recv s
15              in let () = close s
16              in  let [y] = w in y
```

Data availability

No data was used for the research described in the article.

References

[1] K. Honda, Types for dyadic interaction, in: E. Best (Ed.), CONCUR'93, Springer Berlin Heidelberg, Berlin, Heidelberg, 1993, pp. 509–523.
[2] N. Yoshida, V.T. Vasconcelos, Language primitives and type discipline for structured communication-based programming revisited: two systems for higher-order session communication, Electron. Notes Theor. Comput. Sci. 171 (4) (2007) 73–93, https://doi.org/10.1016/j.entcs.2007.02.056.
[3] J.-Y. Girard, Linear logic, Theor. Comput. Sci. 50 (1) (1987) 1–101, https://doi.org/10.1016/0304-3975(87)90045-4.
[4] P. Wadler, Linear types can change the world!, in: M. Broy, C.B. Jones (Eds.), Programming Concepts and Methods: Proceedings of the IFIP Working Group 2.2/2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2–5 April, 1990, North-Holland, Amsterdam, 1990, pp. 561–581, https://homepages.inf.ed.ac.uk/wadler/topics/linear-logic.html#linear-types.
[5] D. Walker, Substructural type systems, in: Advanced Topics in Types and Programming Languages, 2005, pp. 3–44.
[6] L. Caires, F. Pfenning, Session types as intuitionistic linear propositions, in: Proceedings of the 21st International Conference on Concurrency Theory, CONCUR'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 222–236.
[7] B. Toninho, L. Caires, F. Pfenning, A decade of dependent session types, in: N. Veltri, N. Benton, S. Ghilezan (Eds.), PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021, ACM, 2021, pp. 3:1–3:3.
[8] P. Wadler, Propositions as sessions, J. Funct. Program. 24 (2–3) (2014) 384–418, https://doi.org/10.1017/S095679681400001X.
[9] W. Kokke, J.G. Morris, P. Wadler, Towards races in linear logic, in: Coordination Models and Languages: 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings, Springer, 2019, pp. 37–53.
[10] S.J. Gay, V.T. Vasconcelos, Linear type theory for asynchronous session types, J. Funct. Program. 20 (1) (2010) 19–50, https://doi.org/10.1017/S0956796809990268.
[11] S. Lindley, J.G. Morris, A semantics for propositions as sessions, in: European Symposium on Programming Languages and Systems, Springer, 2015, pp. 560–584.
[12] J.-Y. Girard, A. Scedrov, P.J. Scott, Bounded linear logic: a modular approach to polynomial-time computability, Theor. Comput. Sci. 97 (1) (1992) 1–66, https://doi.org/10.1007/978-1-4612-3466-1_11.
[13] A. Brunel, M. Gaboardi, D. Mazza, S. Zdancewic, A core quantitative coeffect calculus, in: Proceedings of the 23rd European Symposium on Programming Languages and Systems, vol. 8410, Springer-Verlag, Berlin, Heidelberg, 2014, pp. 351–370.
[14] D.R. Ghica, A.I. Smith, Bounded linear types in a resource semiring, in: Proceedings of the 23rd European Symposium on Programming Languages and Systems, vol. 8410, Springer-Verlag, Berlin, Heidelberg, 2014, pp. 331–350.
[15] T. Petricek, D. Orchard, A. Mycroft, Coeffects: a calculus of context-dependent computation, in: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, ICFP '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 123–135.
[16] D. Orchard, V.-B. Liepelt, H. Eades III, Quantitative program reasoning with graded modal types, Proc. ACM Program. Lang. 3 (ICFP) (2019) 1–30, https://doi.org/10.1145/3341714.
[17] D. Marshall, D. Orchard, Replicate, reuse, repeat: capturing non-linear communication via session types and graded modal types, in: Carbone and Neykova [46], pp. 1–11, https://doi.org/10.4204/EPTCS.356.1.
[18] P. Choudhury, H. Eades III, R.A. Eisenberg, S. Weirich, A graded dependent type system with a usage-aware semantics, Proc. ACM Program. Lang. 5 (POPL) (2021) 1–32, https://doi.org/10.1145/3434331.
[19] A. Abel, J. Bernardy, A unified view of modalities in type systems, Proc. ACM Program. Lang. 4 (ICFP) (2020) 90:1–90:28, https://doi.org/10.1145/3408972.
[20] J.-P. Bernardy, M. Boespflug, R.R. Newton, S. Peyton Jones, A. Spiwack, Linear Haskell: practical linearity in a higher-order polymorphic language, Proc. ACM Program. Lang. 2 (POPL) (Dec 2017), https://doi.org/10.1145/3158093.
[21] E.C. Brady, Idris 2: quantitative type theory in practice, https://doi.org/10.4230/LIPIcs.ECOOP.2021.9, 2021.
[22] J. Wood, R. Atkey, A framework for substructural type systems, in: I. Sergey (Ed.), Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, in: Lecture Notes in Computer Science, vol. 13240, Springer, 2022, pp. 376–402.
[23] C. McBride, I got plenty o' nuttin', in: S. Lindley, C. McBride, P.W. Trinder, D. Sannella (Eds.), A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, in: Lecture Notes in Computer Science, vol. 9600, Springer, 2016, pp. 207–233.
[24] M. Gaboardi, S.-y. Katsumata, D. Orchard, F. Breuvart, T. Uustalu, Combining effects and coeffects via grading, in: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Association for Computing Machinery, New York, NY, USA, 2016, pp. 476–489.
[25] V. Rajani, D. Garg, Types for information flow control: labeling granularity and semantic models, in: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), 2018, pp. 233–246.
[26] A. Spiwack, C. Kiss, J.-P. Bernardy, N. Wu, R.A. Eisenberg, Linearly qualified types: generic inference for capabilities and uniqueness, Proc. ACM Program. Lang. 6 (ICFP) (Aug 2022), https://doi.org/10.1145/3547626.
[27] B. Toninho, N. Yoshida, Depending on session-typed processes, in: C. Baier, U.D. Lago (Eds.), Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, in: Lecture Notes in Computer Science, vol. 10803, Springer, 2018, pp. 128–145.
[28] B. Moon, H. Eades III, D. Orchard, Graded modal dependent type theory, in: N. Yoshida (Ed.), Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, in: Lecture Notes in Computer Science, vol. 12648, Springer, 2021, pp. 462–490.

[29] M. Berger, K. Honda, N. Yoshida, Sequentiality and the π-calculus, in: International Conference on Typed Lambda Calculi and Applications, Springer, 2001, pp. 29–45.
[30] D. Sangiorgi, D. Walker, The π-Calculus: a Theory of Mobile Processes, Cambridge University Press, 2003.
[31] V.T. Vasconcelos, Fundamentals of session types, Inf. Comput. 217 (2012) 52–70, https://doi.org/10.1016/j.ic.2012.05.002.
[32] S. Peyton Jones, D. Vytiniotis, S. Weirich, G. Washburn, Simple Unification-Based Type Inference for GADTs, ACM SIGPLAN Notices, vol. 41, ACM, 2006, pp. 50–61.
[33] U. Zalakain, O. Dardha, π with leftovers: a mechanisation in agda, in: K. Peters, T.A.C. Willemse (Eds.), Formal Techniques for Distributed Objects, Components, and Systems, Springer International Publishing, Cham, 2021, pp. 157–174.
[34] K. Pruiksma, F. Pfenning, A message-passing interpretation of adjoint logic, Electron. Proc. Theor. Comput. Sci. 291 (2019) 60–79, https://doi.org/10.4204/eptcs.291.6.
[35] B. van den Heuvel, J.A. Pérez, Session type systems based on linear logic: classical versus intuitionistic, PLACES@ ETAPS 314 (2020) 1–11.
[36] S. Balzer, F. Pfenning, Manifest sharing with session types, Proc. ACM Program. Lang. 1 (ICFP) (Aug 2017), https://doi.org/10.1145/3110281.
[37] W. Kokke, O. Dardha, Deadlock-free session types in linear Haskell, in: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell, Haskell 2021, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1–13.
[38] S.J. Fowler, Typed Concurrent Functional Programming with Channels, Actors and Sessions, 2019.
[39] L. Bocchi, M. Murgia, V.T. Vasconcelos, N. Yoshida, Asynchronous timed session types - from duality to time-sensitive processes, in: L. Caires (Ed.), Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, in: Lecture Notes in Computer Science, vol. 11423, Springer, 2019, pp. 583–610.
[40] D. Marshall, D. Orchard, How to take the inverse of a type, in: K. Ali, J. Vitek (Eds.), 36th European Conference on Object-Oriented Programming (ECOOP 2022), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 222, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022, pp. 5:1–5:27.
[41] J. Jacobs, S. Balzer, R. Krebbers, Multiparty GV: functional multiparty session types with certified deadlock freedom, Proc. ACM Program. Lang. 6 (ICFP) (Aug 2022), https://doi.org/10.1145/3547638.
[42] D. Marshall, M. Vollmer, D. Orchard, Linearity and Uniqueness: an Entente Cordiale, in: I. Sergey (Ed.), Programming Languages and Systems, Springer International Publishing, Cham, 2022, pp. 346–375.
[43] D. Marshall, D. Orchard, Functional ownership through fractional uniqueness, Proc. ACM Program. Lang. 8 (OOPSLA1) (Apr 2024), https://doi.org/10.1145/3649848.
[44] E. de Vries, A. Francalanza, M. Hennessy, Uniqueness typing for resource management in message-passing concurrency, J. Log. Comput. 24 (3) (2012) 531–556, https://doi.org/10.1093/logcom/exs022.
[45] P. Rocha, L. Caires, Propositions-as-types and shared state, Proc. ACM Program. Lang. 5 (ICFP) (Aug 2021), https://doi.org/10.1145/3473584.
[46] M. Carbone, R. Neykova (Eds.), Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2022, Munich, Germany, 3rd April 2022, EPTCS, vol. 356, 2022.
[47] J. Hughes, D. Marshall, J. Wood, D. Orchard, Linear exponentials as graded modal types, in: 5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021), Rome (Virtual), Italy, 2021, https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271465.