Replicate, Reuse, Repeat: Capturing Non-Linear Communication via Session Types and Graded Modal Types

Daniel Marshall
University of Kent, UK
dm635@kent.ac.uk

Dominic Orchard
University of Kent & University of Cambridge, UK
d.a.orchard@kent.ac.uk

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, a strictly linear setting is somewhat limiting, since there exist various useful patterns of communication that rely on non-linear behaviours. For example, shared channels provide a way to repeatedly spawn a process with binary communication along a fresh linear session-typed channel. Non-linearity can be introduced in a controlled way in programming through the general concept of *graded modal types*, which are a framework encompassing various kinds of *coeffect* typing (describing how computations make demands on their context). This paper shows how graded modal types can be leveraged alongside session types to enable various non-linear concurrency behaviours to be re-introduced in a precise manner in a type system with a linear basis. The ideas here are demonstrated using Granule, a functional programming language with linear, indexed, and graded modal types.

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. *Behavioural types* on the other hand capture not just *what* data is, but *how* it is calculated. One such behavioural type system for concurrency is *session types* [10], representing the behaviour of a process communicating over a channel via the channel's type. Session types naturally fit into the more general substructural discipline of *linear types* [8, 23, 25].

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 and must be used according to a 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 and classical linear logic [4, 24]. Furthermore, linearly-typed functional languages then provide an excellent basis for session-typed programming [6].

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 [9]. 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* [3, 7, 17], which are a framework for capturing different kinds of resource analysis in a single type system. *Graded modal types* [16] unify coeffects with their dual notion of *effects*, and are an expressive system allowing for the specification and verification of many behavioural properties of programs. In this paper, we demonstrate that combining session types with graded modal types allows for various non-linear behaviours of concurrent programs to be reintroduced in a linear setting, in a controlled and precise way.

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

M. Carbone and R. Neykova (Eds.): 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software
EPTCS 356, 2022, pp. 1–11, doi:10.4204/EPTCS.356.1

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) provides unsoundness. This is discussed in more detail in Section 4, where we also describe our chosen solution. 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*).

Related ideas appear in the type system of Zalakain and Dardha who use leftover typing to define a resource-aware session type calculus that can represent shared, graded and linear channels [27]. We capture some of the same ideas via a unified graded approach, showing how different amounts of sharing can be characterised precisely with the interaction of linear, indexed, and graded types.

2 A Brief Granule and Graded Modal Types Primer

Granule's type system is based on the linear λ-calculus augmented with *graded modal types* [16]. 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 code:

```
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), 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 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 ! [8] and Bounded Linear Logic [9]. The above example uses the natural number graded modality, counting 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 for retrieving a value that may or may not exist 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:

```
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.

Type system redux The core type theory (based on Orchard et al. [16]) extends the linear λ-calculus with the semiring-graded necessity modality $\Box_r A$. Typing contexts contain both linear assumptions $x : A$ and graded assumptions $x : [A]_r$ which have originated from inside a graded modality. The core typing rules for introducing and eliminating graded modal types are then as follows:

$$\frac{\Gamma \vdash t : A}{r * \Gamma \vdash [t] : \Box_r A}\text{PR} \quad \frac{\Gamma \vdash t_1 : \Box_r A \qquad \Delta, x : [A]_r \vdash t_2 : B}{\Gamma + \Delta \vdash \textbf{let } x = t_1 \textbf{ in } t_2 : B}\text{ELIM} \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\text{DER} \quad \frac{\Gamma \vdash t : B}{\Gamma, 0 * \Delta \vdash t : B}\text{WEAK}$$

The PR rule (promotion) introduces a graded modality with grade r, and thus must scale by r all of the inputs (none of which are allowed to be linear as $r * \Gamma$ is partial: it scales each graded assumption in Γ by r or it is undefined if Γ contains linear assumptions). The ELIM rule 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 as given by the 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 connects linear typing to graded typing, showing that a requirement for a linear assumption is satisfied by an assumption graded by 1. Lastly, WEAK explains how we can *weaken* with variables graded by 0.

Implicit in any rule involving multiple terms is a notion of *contraction* captured by the $+$ operation on contexts, which is only defined when contexts are disjoint with respect to linear assumptions, and on overlapping graded assumptions we add their grades, e.g. $(\Gamma, x : [A]_r) + (\Delta, x : [A]_s) = (\Gamma + \Delta), x : [A]_{r+s}$.

3 Linear Session Types in Granule

The implementation of session types in Granule is based on the GV calculus (originating from Gay and Vasconcelos [6], further developed by Wadler [24], for which we use Lindley and Morris' formu-

lation [12]). The GV system extends the linear λ-calculus with a data type of channels $\mathsf{Chan}(S)$ parameterised by session types S [26], which capture the protocol of interaction allowed over the channel. Communication is asynchronous (sending always succeeds, but receiving can block as usual).

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:

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

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 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 → ()
```

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 [12] these can be defined in terms of the core functions given above, but we provide primitives 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:

```
1  server : LChan (Offer (Recv Int End) (Recv () End)) → Int [0..1] → Int
2  server c [d] = offer (λc → let (x, c) = recv c in let () = close c in x)
3                       (λc → let ((), c) = recv c in let () = close c in d) c
4
5  client : ∀ {p : Protocol} . LChan (Select (Send Int End) p) → ()
6  client c = let c = selectLeft c;
7                 c = send c 42
8             in close c
9
10 example : Int -- Evaluates to 42
11 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 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.

4 The Relationship Between Grading, Call-by-Value, and Effects

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 [3, 5]. 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 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).[1] 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 linearity.

Our alternative solution is that we instead simply syntactically restrict promotion for primitives which create linear channels; in particular this includes the `forkLinear` function which causes the difficulty here. Non-linear channels are then re-introduced by additional primitives in Section 5 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.

5 Non-linear Communication Patterns via Grading

We show three common patterns and how they can be described by combining linear session types and graded modal types: *reusable channels* (Section 5.1), *replicated servers* (Section 5.2), and *multicast communication* (Section 5.3). All of these patterns are a variation on the `forkLinear` primitive, with some amount of substructural behaviour introduced via graded modal types, and sometimes restrictions to pro-

[1] 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`.

tocols via *type predicates*. In Granule these are represented similarly to Haskell's type class constraints with type signatures of the form: `functionName : ∀ {typeVariables} . {constraints} ⇒ type`.

5.1 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 issue 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 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}
              ⇒ ((LChan p) [r] → ()) → (LChan (Dual p)) [r]
```

where `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) = let () = close (send c x) 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; () = close c' in Cons x (recvVec n [c])
4
5  example : ∀ {n : Nat, a : Type} . Vec n a → Vec n a
6  example xs = let (n, list) = length' list
7               in recvVec n (forkNonLinear (λc → sendVec c list))
8
9  main : Vec 5 Int
10 main = example (Cons 1 (Cons 1 (Cons 2 (Cons 3 (Cons 5 Nil)))))
```

Note that the receiver needs to know how many elements to receive, so this information has to be passed separately (via an indexed natural number `N n`). A system with dependent session types [20, 21] 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 [15].

5.2 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 [1, 19], 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.

We provide this 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 5.1; 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, which includes both `ReceivePrefix (Recv a p)` and `ReceivePrefix (Offer p1 p2)`.

Each 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. 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.

The following example demonstrates `forkReplicate` in action with two clients.

```
1  addServer : LChan (Offer (Recv Int (Recv Int (Send Int End))) (Recv Int (Send Bool End))) → ()
2  addServer c =
3   offer (λc → let (x, c) = recv c;  (y, c) = recv c; in close (send c (x + y))
4         (λc → let (x, c) = recv c;                   in close (send c (x == 0))) c
5
6  client1 : ∀ {p : Protocol} . LChan (Select (Send Int (Send Int (Recv Int End))) p) → Int
7  client1 c = let (x, c) = recv (send (send (selectLeft c) 10) 20); () = close c in x
8
9  client2 : ∀ {p : Protocol} . LChan (Select p (Send Int (Recv Bool End))) → Bool
10 client2 c = let (b, c) = recv (send (selectRight c) 42);  () = close c in b
11
12 import Parallel -- Provides a `par` combinator derived from `forkLinear`
13 main : (Int, Bool)
14 main = let (Cons [c1] (Cons [c2] Nil)) = forkReplicate [addServer] (S (S Z))
15        in par (λ() → client1 c1) (λ() → client2 c2)
```

Here `addServer` provides functionality offering the ability to receive two integers and send back their addition, or receive a single integer and send back whether it is a zero or not. The two clients `client1` and `client2` exercise both behaviours, via channels given by `forkReplicate` of `addServer`, and are run in parallel on line 15 using the `par` combinator which itself implemented in terms of `forkLinear`.

In the intuitionistic linear logical propositions of Caires and Pfenning [4], this same idea 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* [26]. Our approach codifies the same principle using graded modalities, but only allows finite replication (Section 6 discusses unbounded replication).

5.3 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 clients. The non-linearity here is now on the payload values being sent, with grading to explain that the amount of non-linearity matches the number of clients. The `forkMulticast` primitive provides this behaviour:

```
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 type function adding a graded modality to payload types:

```
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)

Graded n End = End
```

and `Sends : Protocol → Predicate` is defined:

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

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 participants via the `Vec n (Chan (Dual p))` channels.

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.

```
1 broadcaster : LChan (Send (Int [3]) End) → ()
2 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`.

```
1 aggregateRecv : ∀ {n : Nat} . Vec n (LChan (Recv Int End)) → Int
2 aggregateRecv Nil         = 0;
3 aggregateRecv (Cons c cs) = let (x, c) = recv c; () = close c in x + aggregateRecv cs
4
5 main : Int -- Evaluates to 126
6 main = aggregateRecv (forkMulticast broadcaster (S (S (S Z))))
```

6 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. This differs, but is related to, the generalisation of session types via adjoint logic presented by Pruiksma and Pfenning [18], which offers non-linearity but without the precise quantification allowed by our grades. The demands of call-by-value required a different approach to past work, with syntactic restriction discussed in Section 4 and vectors to capture multiplicity of channels. The next step would be to prove type safety given the additional restriction on promoting channels and the other features introduced here.

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 5.1) are equivalent to linearly recursive session types (e.g., $\mu x.P.x$), especially when the grade `r` is instantiated to `0..∞` to capture an arbitrary amount of use. Further work is to integrate standard ideas on recursive session types and to explore their interaction with grading.

Combinations of the ideas here are also possible; for example, combining multicast sending with reuse to get channels which we can repeatedly use to broadcast upon. We could also define a more powerful version of replication to allow 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:

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

Further applications and related ideas Using the linear channels already present in Granule it is possible to represent functions that are *sequentially realizable* [13]; 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.

As discussed in Section 4, it is necessary to restrict promotion of channels due to Granule's default call-by-value semantics, since otherwise this 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 [14]; if a unique array is promoted then we end up with multiple references to the same array, so we can no longer guarantee uniqueness for mutation. This is resolved by restricting promotion in much the same way as we have here for linear channels.

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 [22]. 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.

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 [2], 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 [11], 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.

Acknowledgments This work was supported by an EPSRC Doctoral Training Award (Marshall) and EPSRC grant EP/T013516/1 (*Verifying Resource-like Data Use in Programs via Types*).

References

[1] Martin Berger, Kohei Honda & Nobuko Yoshida (2001): *Sequentiality and the π-calculus*. In: *International Conference on Typed Lambda Calculi and Applications*, Springer, pp. 29–45, doi:10.1007/3-540-45413-6_7.

[2] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones & Arnaud Spiwack (2017): *Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language*. *Proc. ACM Program. Lang.* 2(POPL), doi:10.1145/3158093.

[3] Aloïs Brunel, Marco Gaboardi, Damiano Mazza & Steve Zdancewic (2014): *A Core Quantitative Coeffect Calculus*. In: *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, Springer-Verlag, Berlin, Heidelberg, p. 351–370, doi:10.1007/978-3-642-54833-8_19.

[4] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In: *Proceedings of the 21st International Conference on Concurrency Theory*, CONCUR'10, Springer-Verlag, Berlin, Heidelberg, p. 222–236, doi:10.1007/978-3-642-15375-4_16.

[5] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart & Tarmo Uustalu (2016): *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, p. 476–489, doi:10.1145/2951913.2951939.

[6] Simon J. Gay & Vasco Thudichum Vasconcelos (2010): *Linear type theory for asynchronous session types*. *J. Funct. Program.* 20(1), pp. 19–50, doi:10.1017/S0956796809990268.

[7] Dan R. Ghica & Alex I. Smith (2014): *Bounded Linear Types in a Resource Semiring*. In: *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, Springer-Verlag, Berlin, Heidelberg, p. 331–350, doi:10.1007/978-3-642-54833-8_18.

[8] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50(1), pp. 1 – 101, doi:10.1016/0304-3975(87)90045-4. Available at http://www.sciencedirect.com/science/article/pii/0304397587900454.

[9] Jean-Yves Girard, Andre Scedrov & Philip J Scott (1992): *Bounded linear logic: a modular approach to polynomial-time computability*. *Theoretical Computer Science* 97(1), pp. 1–66, doi:10.1007/978-1-4612-3466-1_11.

[10] Kohei Honda (1993): *Types for dyadic interaction*. In Eike Best, editor: *CONCUR'93*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[11] Wen Kokke & Ornela Dardha (2021): *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, p. 1–13, doi:10.1145/3471874.3472979.

[12] Sam Lindley & J Garrett Morris (2015): *A semantics for propositions as sessions*. In: *European Symposium on Programming Languages and Systems*, Springer, pp. 560–584, doi:10.1007/978-3-662-46669-8_23.

[13] Daniel Marshall & Dominic Orchard (2022): *How to Take the Inverse of a Type*. To appear in ECOOP 2022.

[14] Daniel Marshall, Michael Vollmer & Dominic Orchard (2022): *Linearity and Uniqueness: An Entente Cordiale*. To appear in ESOP 2022.

[15] Benjamin Moon, Harley Eades III & Dominic Orchard (2021): *Graded Modal Dependent Type Theory*. In Nobuko Yoshida, editor: *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*, *Lecture Notes in Computer Science* 12648, Springer, pp. 462–490, doi:10.1007/978-3-030-72019-3_17.

[16] Dominic Orchard, Vilem-Benjamin Liepelt & Harley Eades III (2019): *Quantitative program reasoning with graded modal types*. *Proceedings of the ACM on Programming Languages* 3(ICFP), pp. 1–30, doi:10.1145/3341714.

[17] Tomas Petricek, Dominic Orchard & Alan Mycroft (2014): *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, p. 123–135, doi:10.1145/2628136.2628160.

[18] Klaas Pruiksma & Frank Pfenning (2019): *A Message-Passing Interpretation of Adjoint Logic*. *Electronic Proceedings in Theoretical Computer Science* 291, p. 60–79, doi:10.4204/eptcs.291.6.

[19] Davide Sangiorgi & David Walker (2003): *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press.

[20] Bernardo Toninho, Luís Caires & Frank Pfenning (2021): *A Decade of Dependent Session Types*. In Niccolò Veltri, Nick Benton & Silvia Ghilezan, editors: *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*, ACM, pp. 3:1–3:3, doi:10.1145/3479394.3479398.

[21] Bernardo Toninho & Nobuko Yoshida (2018): *Depending on Session-Typed Processes*. In Christel Baier & Ugo Dal Lago, editors: *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*, *Lecture Notes in Computer Science* 10803, Springer, pp. 128–145, doi:10.1007/978-3-319-89366-2_7.

[22] Edsko de Vries, Adrian Francalanza & Matthew Hennessy (2012): *Uniqueness typing for resource management in message-passing concurrency*. *Journal of Logic and Computation* 24(3), pp. 531–556, doi:10.1093/logcom/exs022. arXiv:https://academic.oup.com/logcom/article-pdf/24/3/531/2782308/exs022.pdf.

[23] Philip Wadler (1990): *Linear Types Can Change the World!* In Manfred Broy & Cliff B. Jones, editors: *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, pp. 561–581. Available at https://homepages.inf.ed.ac.uk/wadler/topics/linear-logic.html#linear-types.

[24] Philip Wadler (2014): *Propositions as sessions*. *Journal of Functional Programming* 24(2-3), pp. 384–418, doi:10.1017/S095679681400001X.

[25] David Walker (2005): *Substructural Type Systems*. *Advanced Topics in Types and Programming Languages*, pp. 3–44.

[26] Nobuko Yoshida & Vasco T Vasconcelos (2007): *Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication*. *Electronic Notes in Theoretical Computer Science* 171(4), pp. 73–93, doi:10.1016/j.entcs.2007.02.056.

[27] Uma Zalakain & Ornela Dardha (2021): *π with Leftovers: A Mechanisation in Agda*. In Kirstin Peters & Tim A. C. Willemse, editors: *Formal Techniques for Distributed Objects, Components, and Systems*, Springer International Publishing, Cham, pp. 157–174, doi:10.1007/978-3-030-78089-0_9.