How to Take the Inverse of a Type

Daniel Marshall
School of Computing, University of Kent, Canterbury, UK

Dominic Orchard
School of Computing, University of Kent, Canterbury, UK
Department of Computer Science and Technology, University of Cambridge, UK

Abstract

In functional programming, regular types are a subset of algebraic data types formed from products and sums with their respective units. One can view regular types as forming a commutative semiring but where the usual axioms are isomorphisms rather than equalities. In this pearl, we show that regular types in a *linear* setting permit a useful notion of *multiplicative inverse*, allowing us to "divide" one type by another. Our adventure begins with an exploration of the properties and applications of this construction, visiting various topics from the literature including program calculation, Laurent polynomials, and derivatives of data types. Examples are given throughout using Haskell's linear types extension to demonstrate the ideas. We then step through the looking glass to discover what might be possible in richer settings; the functional language Granule offers linear functions that incorporate local side effects, which allow us to demonstrate further algebraic structure. Lastly, we discuss whether dualities in linear logic might permit the related notion of an *additive inverse*.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases linear types, regular types, algebra of programming, derivatives

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.5

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.1`

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

Acknowledgements Thanks to Nicolas Wu and Harley Eades III for their valuable comments and discussion on earlier drafts, and also to the anonymous reviewers for their helpful feedback.

1 Prologue: Consuming with Inverses

Algebraic data types are the bread-and-butter of both the theory and practice of functional programming. The algebraic view gives rise to vast possibilities for manipulating types, and for "calculating" programs from their type structure in the Bird-Meertens tradition [7, 8, 37].

Regular types are a subset of algebraic types formed from products $\times$, sums $+$, unit 1 and empty types 0, and fixed points, giving rise to polynomial type expressions [37, 40] and an algebraic structure akin to a commutative semiring. The multiplicative part is by products and the unit type, and the additive part by sums and the empty type. However, the semiring laws are relaxed to isomorphisms, e.g., $a \times (b \times c) \cong (a \times b) \times c$ is witnessed by a bijection between the two ways of associating a triple expressed as pairs. The cardinality operation $|-|$ (mapping a type to its size) is then a semiring homomorphism (a functor) from the structure of types to natural numbers, e.g., $|a \times b| = |a||b|$. This provides a useful technique for understanding when different type expressions are isomorphic by checking if their cardinalities are equal, a fact leveraged by many a student for decades. In category theory, we can model regular types as a *(commutative) semiring category* (or *rig category* [11]) or a *(symmetric) bimonoidal category* [26], with semiring rules as natural isomorphisms. Either way, a rose by any other name is still as sweet.

Given the rich algebraic analogues and models for types, one may then (perhaps idly) wonder: if product types are multiplicative, *is there a notion of multiplicative inverse for types which lets us divide one type by another*? We show that this question has a rather neat and simple answer for *linear types*:

▶ **Definition 1.** The multiplicative inverse of type τ is the type of functions *consuming* τ:

$$\tau^{-1} \triangleq \tau \multimap 1$$

where $\multimap$ is the linear function type, of functions which use their argument exactly once.

Linear types are an ideal setting for capturing the idea of *consumption* since linear types treat values as resources which must be used exactly once – they can never be discarded (no weakening) or duplicated or shared (no contraction) [20, 51, 55]. Therefore, a function $\tau \multimap 1$ must consume its argument rather than simply returning a value of the unit type.

Linear regular types have products $\otimes$ ("multiplicative conjunction") where each component of the pair is used exactly once and sums $\oplus$ ("additive disjunction") which behave like normal sum types, though whichever component we are given must of course be used linearly. Linear regular types again form a commutative semiring structure, with equalities as isomorphisms.

Given $\tau^{-1} \triangleq \tau \multimap 1$, we can immediately consult the standard meaning of multiplicative inverse with regards to its equational theory: a group extends a monoid $(X, \bullet, e)$ such that for every $x \in X$ there exists an *inverse* element denoted $x^{-1} \in X$, for which $x \bullet x^{-1} = e$ and $x^{-1} \bullet x = e$. If the use of the notation $-^{-1}$ and the terminology of inverses is warranted then we could reasonably expect $\tau^{-1} \otimes \tau \cong 1$. Given the above definition of inverse, then one direction of this isomorphism, from $\tau^{-1} \otimes \tau$ to 1, is inhabited via function application:

$$\lambda(u, x).\ u\ x\ :\ (\tau^{-1} \otimes \tau) \multimap 1 \tag{1}$$

where the first component of the pair consumes the second component. Similarly we can construct the symmetric version $(\tau \otimes \tau^{-1}) \multimap 1$ by first flipping the components of the pair.

These are the only inhabitants of their types: the multiplicative inverse of τ *must* consume the τ value in the other component of the pair due to the constraints of linearity. Indeed, starting from the goal of defining a multiplicative inverse for a linear type such that the property $\tau^{-1} \otimes \tau \multimap 1$ holds, by "currying" we must have a map from τ^{-1} to $\tau \multimap 1$. Therefore, $\tau \multimap 1$ is the most natural choice for an inverse to τ, as any other inverse would first need to be mapped to $\tau \multimap 1$ in order to consume the value of type τ.

We call equation (1) and its symmetric counterpart *lax inverse laws* following category theory terminology: *strict* structures have equalities, *strong* have isomorphisms, and *lax* have only morphisms in one direction. The rest of the commutative semiring structure of linear regular types is "strong" as associativity, commutativity, etc. are isomorphisms. However, for inverses, the opposite direction from 1 to $\tau^{-1} \otimes \tau$ does not exist in general: for any τ we cannot necessarily form a pair $\tau^{-1} \otimes \tau$ as we do not have an algorithm to construct an arbitrary τ value nor its consumer. Even if we can inhabit $1 \multimap (\tau^{-1} \otimes \tau)$ for a particular τ (e.g., if there is a default value for a type and a standard way of consuming its values) this only forms an isomorphism $\tau^{-1} \otimes \tau \cong 1$ in the limited setting of 1-element types. The crux is that division *loses* information: $\tau^{-1} \otimes \tau \multimap 1$ consumes knowledge of the original τ value.

In a non-linear ("Cartesian") setting, we could still define inverses in a similar way as $\tau^{-1} \triangleq \tau \to 1$, and the lax inverse law $\tau^{-1} \times \tau \to 1$ would be similarly inhabited by function application. However, non-linearity makes this definition weaker and less natural, as $((\tau \to 1) \times \tau) \to 1$ is also inhabited by the function $\lambda(u, x).1$ which throws away both of its arguments and returns the unit value. We could just as easily construct this term of

type $(\tau^{-1} \times \tau) \to 1$ regardless of the definition we chose for τ^{-1}. It is only by working in the context of linear types that our notion of inverse is given meaning, as a function $\tau \otimes \tau^{-1} \multimap 1$ must use both the term τ and its inverse, for which $\tau^{-1} \triangleq \tau \multimap 1$ is the natural fit.

This definition of τ^{-1} lets us view types as an algebraic structure which is *almost* a semifield; a semifield resembles a semiring except that every nonzero element has a multiplicative inverse. The terminology of *skewness* can also be applied here; for example, a *skew monoid* is one in which the associativity and unit properties are morphisms rather than isomorphisms or equalities [48, 50]. Thus our construction could be described as a "multiplicative skew inverse". Going forwards we use just "inverse" for brevity.

Roadmap

In this pearl, we explore various applications and consequences of this idea. We begin by programming with inverses in Haskell via the linear types extension of the Glasgow Haskell Compiler[1] (based on the work of Bernardy et al. [6]), and proceed to consider equations arising from functions over inverses and other algebraic implications.

One interesting result we uncover is that whilst regular types yield polynomial type expressions, inverse types yield an analogue of the mathematical generalisation of *Laurent polynomials* (Section 4); these differ from ordinary polynomials in that they can have terms of negative degree, providing a more general notion of exponent for regular types. But inverses turn out to have applications beyond the merely theoretical; we show inverses allow the notion of derivatives for regular types (à la McBride [37]) to be generalized, providing the ability to take the derivative of a type *with respect to another type* (Section 5). This yields a way to generate data types with n-holes (holes of n contiguous elements) which we apply to the common programming idiom of stencil computations.

In the second half, we consider possibilities for developing the algebraic structure of inverse types further, by working in richer and more expressive settings. It turns out that the multiplicative inverse becomes an involution (Section 6) if we can express *sequentially-realizable functions* [32] which carry out local side effects that are not observable externally. We demonstrate this using linear session channels à la Lindley and Morris [30] which allow inverses to do more computationally. We show examples in the modern functional language Granule which has linear types at its core [42]. This involution also happens to yield a construction akin to the familiar continuation monad, which we briefly discuss.

Lastly, we show it is also possible to define an additive inverse (Section 7). However, this requires working in a different setting where products are given by linear logic's & rather than $\otimes$ and similarly sums are $⅋$ rather than $\oplus$. Thus, while we can develop the theory for each kind of inverse separately, there is not yet any type theory where the two can coexist.

A side aim of this pearl is to popularise the increasing abilities of modern functional languages to express linear types. For those unfamiliar with linear types and wishing to go beyond the intuitions here, Appendix A gives some standard typing rules and syntax. We also provide an artifact[2] including all code examples given throughout in both Haskell and Granule, to aid with understanding and allow for further experimentation.

[1] Available as of GHC 9.0.1, `https://www.haskell.org/ghc/download_ghc_9_0_1.html`, released Feb 2021.

[2] `https://doi.org/10.5281/zenodo.6275280`

2 Programming with Inverses

We stand at an exciting juncture for our community. Finally, more than 30 years after their conception in logic [20], linear types are starting to gain a foothold in mainstream functional programming languages. One such language is Haskell, via GHC's linear types extension [6] which uses a graded type system [42] based on annotating function types `a %r -> b` with their "multiplicity" `r` (which can also be understood as a *coeffect*, or *consumption effect* [43]), that describes how many times the argument is used. In Haskell, this can either be `1` representing *linear* behaviour or `Many` representing *unrestricted* behaviour, including the possibility of 0 uses. We can thus describe inverses and a curried version of the lax inverse law as follows:[3]

```
type Inverse a = a %1 -> ()      -- recall () is the unit type 1 of Haskell

divide :: a %1 -> Inverse a %1 -> ()
divide x u = u x
```

The naming of `divide` is to evoke the usual intuition associated with groups where $a/b = a \bullet b^{-1}$ and since this function "actions" the consumption of the first input by the second.

There are other linearly-typed languages in which one could also readily apply our notion of inverses, e.g., ATS [47], Alms [49], and Quill [39]. Through some translation we can also represent inverses in languages with more expressive graded type systems, such as Granule [42] and Idris 2 [10], that can describe linearity as well as other flavours of resourceful data. We focus on Haskell for now, but the ideas are the same no matter the language.

In the concrete setting of an actual language, we can now give an example inhabitant of an inverse type. These are typically defined by some pattern matching over all the possible inputs, where the act of pattern matching on the incoming value consumes the input as it inspect its value. For example, an inverse to Haskell's boolean type is given by:

```
boolDrop :: Inverse Bool
boolDrop True  = ()
boolDrop False = ()
```

The `linear-base` library for Haskell provides a type class for those types which are "consumable": inhabitants of the inverse of type `a`. The instance of `Consumable` for the boolean type is the `boolDrop` function defined explicitly above.

```
class Consumable a where
   consume :: a %1-> ()

instance Consumable Bool where
  consume True  = ()
  consume False = ()
```

Various built-in types like `Int` have a "linearly unsafe" implementation which simply drops the argument rather than, say, consuming a machine integer by matching on the `0` case and otherwise recursively consuming the integer decremented by `1`, which would be safe but slow! This explicit weakening operation can also be algorithmically generated from a regular type, following a generic deriving mechanism [23].

[3] Note that we first need to enable the linear types extension, by using the pragma `{-# LANGUAGE LinearTypes #-}`; this will be left implicit in all of the snippets of Haskell throughout the pearl.

A key aspect of this typing discipline is that we do not want certain types to be consumable without side effects; for example, file handles, sockets, channels, or any other piece of data which acts as a proxy for a resource for which there exists some protocol of interaction. In Section 6, we see more interesting inhabitants of inverse types in a more expressive setting.

We can consider algebraic properties of inverses and understand them through the lens of linear regular types using this definition, while bearing in mind that our inverses are lax, and so the properties will hold only in one direction. For example, consider the following property, which is a simple application of the distributivity of multiplication over addition.

$$\begin{aligned}(\tau \oplus 1) \otimes \tau^{-1} &\cong ((\tau \otimes \tau^{-1}) \oplus \tau^{-1}) \\ &\multimap 1 \oplus \tau^{-1}\end{aligned} \tag{2}$$

We can understand $\tau \oplus 1$ as the linear version of the traditional Haskell `Maybe` data type (called *option* in ML), and thus recover the following function definition in Haskell corresponding to the above (in)equation, giving us a way to distribute an inverse into a `Maybe` value.

```
maybeNeg :: (Maybe a) %1 -> Inverse a %1 -> Maybe (Inverse a)
maybeNeg Nothing  u = Just u
maybeNeg (Just n) u = letUnit (divide n u) Nothing

letUnit :: () %1 -> a %1 -> a -- Abstracts 'let () = t1 in t2' - needed since
letUnit () x = x              -- let bindings are currently always non-linear.
```

In the second case of `maybeNeg`, we cannot simply return `Nothing` since `u` and `n` are linearly typed; we must first apply `u` to `n` (via `divide`) to consume both values. We then want `let () = divide n u in Nothing`, but linear let-bindings are not yet implemented (as of GHC 9.2.2, released in March 2022), so we abstract this pattern as the function `letUnit` instead.

3 Calculating with Inverses

Regular types come equipped with various equations governing their operations which can be used for reasoning about functional programs [18] and even deriving implementations starting from equational specifications (the *Bird-Meertens* formalism) [7, 8, 19]. We consider here analogous equations for calculating with inverses. We explore these equations from the perspective of the linear λ-calculus with regular types, illustrating some points using Haskell for convenience. One can freely translate between the two.

In a linear types setting, many of the usual equations governing products are not available to us, because the "tupling" that combines regular functions $f : A \to B$ and $g : A \to C$ into $\langle f, g \rangle : A \to (B \times C)$ violates linearity by copying a value of type A, and projections $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ violate linearity by discarding one component of a pair. We can however work with $\otimes$ as a bifunctor (which lifts $f : A \multimap B$ and $g : C \multimap D$ to $f \otimes g : A \otimes C \multimap B \otimes D$), and cotupling $[h, k] : A \oplus B \multimap C$ (for $h : A \multimap C$ and $k : B \multimap C$) is still available. Thus we have equations for (bi)functoriality of $\otimes$ and $\oplus$:

$$\begin{aligned} id \otimes id &= id \\ (f \otimes g) \circ (h \otimes k) &= (f \circ h) \otimes (g \circ k) \end{aligned} \qquad \begin{aligned} id \oplus id &= id \\ (f \oplus g) \circ (h \oplus k) &= (f \circ h) \oplus (g \circ k) \end{aligned}$$

and equations interacting cotupling, injections and the $\oplus$ bifunctor, e.g., to name a few:

$$[f, g] \circ \mathsf{inl} = f \qquad [f, g] \circ \mathsf{inr} = g \qquad [h \circ \mathsf{inl}, h \circ \mathsf{inr}] = h$$

For brevity we elide the rest as they are not the object here. Appendix A.1 gives the remaining equations (which are the subset of those from Gibbons [18] that are permitted in a linear setting). There are various other equations arising from the isomorphisms of regular types (Section 1), e.g., for the isomorphisms witnessing associativity with $\alpha : (A \otimes B) \otimes C \multimap A \otimes (B \otimes C)$ and α_i is its converse, then we have equations $\alpha \circ \alpha_i = \alpha_i \circ \alpha = id$.

Inverse as a functor

A few equations arise from the simple fact that multiplicative inverse is a contravariant functor, and thus we have a contravariant "map" function via function composition:

```
comap :: (b %1 -> a) %1 -> Inverse a %1 -> Inverse b
comap f g = \x -> g (f x)
```

A functor's action on a morphism is commonly written using the same symbol as its action on objects (types), just as seen above for $\otimes$ and $\oplus$. However, writing `comap` applied to f as f^{-1} gives the wrong impression: we are not representing the inverse of a function, but rather lifting a function to work on inverses. We therefore write $f^{\ominus 1}$ for `comap` f to avoid confusion.[4]

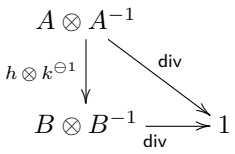
We therefore have the functoriality equations for inverses:

$$id^{\ominus 1} = id \qquad g^{\ominus 1} \circ f^{\ominus 1} = (f \circ g)^{\ominus 1}$$

Let $\mathsf{div} : A \otimes A^{-1} \multimap 1$ be the lax inverse law as a function in the linear λ-calculus (the uncurried form of the function `divide :: a -> Inverse a -> ()` we defined for Haskell earlier). Given two functions $h : A \multimap B, k : B \multimap A$, we then have the following naturality property:

$$\mathsf{div} \circ (h \otimes k^{\ominus 1}) = \mathsf{div}$$

which can be seen more clearly in a diagram as:



i.e., transforming a value and its inverse prior to consumption is the same as us just consuming the original value. This property follows from the definitions. We revisit this law in Section 6 once we introduce consuming functions that can have some (safe-by-linearity) side effect.

Monoidal structure of inverses

The inverse (contravariant) functor also has additional monoidal functor structure, which we can write in Haskell simply as:

```
munit :: () %1 -> Inverse ()
munit () = (\() -> ())

mmult :: (Inverse a) %1 -> (Inverse b) %1 -> Inverse (a, b)
mmult f g = \(a, b) -> letUnit (f a) (g b)
```

[4] We considered using the notation $\tau^{\ominus 1}$ for types as well, but thought it was too ugly to put everywhere.

i.e., `munit` consumes a unit value by pattern matching then returning unit (the standard polymorphic identity function would have worked equally well), and `mmult` returns a function that consumes a pair by using `f` to consume the first component then `g` to consume the second. The usual axioms of a (lax) monoidal functor (associativity and unit laws) [34] hold, interacting with the monoidal structure of $\otimes$; we detail these axioms in Appendix A.1.

This monoidal functor structure on inverses gives us the simple idea that we can combine multiple inverses into a composite inverse; in other words, a pair of consumers can be turned into a consumer of pairs. This satisfies the following equation:

$$\rho \circ (\mathsf{div} \otimes \mathsf{div}) = \mathsf{div} \circ (id \otimes \texttt{mmult}) \circ \mathit{interchange}$$

where $\rho : 1 \otimes 1 \multimap 1$ collapses units and $\mathit{interchange} : (A \otimes B) \otimes (C \otimes D) \multimap (A \otimes C) \otimes (B \otimes D)$ is derived from associativity and commutativity isomorphisms. This rule can be seen more clearly as a diagram:

$$\begin{array}{ccccc} (A \otimes A^{-1}) \otimes (B \otimes B)^{-1} & & \xrightarrow{\mathit{interchange}} & & (A \otimes B) \otimes (A^{-1} \otimes B^{-1}) \\ \Big\downarrow{\scriptstyle \mathsf{div} \otimes \mathsf{div}} & & & & \Big\downarrow{\scriptstyle id \otimes \texttt{mmult}} \\ 1 \otimes 1 & & \xrightarrow[\rho]{} & 1 & \xleftarrow[\mathsf{div}]{} (A \otimes B) \otimes (A \otimes B)^{-1} \end{array}$$

i.e., we can perform two inverse eliminations or we can rearrange, combine the inverses into one, and then apply a single elimination.

The idea of combining products of inverses together leads naturally to notions of *exponentiation*, but with negative exponents, which we explore next.

4 Exponentiation with Inverses

In the standard semiring of (linear) regular types discussed in Section 1, the type constructors 0, 1, $\otimes$ and $\oplus$ generate polynomials over some type meta-variable where terms with exponents τ^n are represented by the n-wise product of τ where $\tau^0 = 1$. For $n \geq 0$, this gives us the usual positive exponent laws up to isomorphism (using associativity and commutativity):

$$\forall a, b.\, (a \geq 0 \wedge b \geq 0) \quad \tau^a \otimes \tau^b \cong \tau^{a+b} \qquad (\exp_+)$$

$$\forall a.\, (a \geq 0) \quad \sigma^a \otimes \tau^a \cong (\sigma \otimes \tau)^a \qquad (\exp_\sigma)$$

Introducing the multiplicative inverse allows us to generalise these to negative exponents, and thus to generate *Laurent polynomials* over types, which differ from ordinary polynomials in that they can have terms of negative degree [28].

We define exponentiation over a type τ for negative exponents as

$$\forall a.(a \geq 0) \qquad \tau^{-a} \triangleq (\tau^{-1})^a$$

For example, $\tau^{-2} = (\tau^{-1})^2 = \tau^{-1} \otimes \tau^{-1}$ capturing a pair of inverses.

It is clear that the first exponential law (equation $\exp_+$) generalises in the case that both coefficients are negative, since we define τ^{-n} as a product of n inverses τ^{-1} in much the same way that τ^n represents a product of n values of type τ, i.e.,

$$\forall a, b.\, (a \geq 0 \wedge b \geq 0) \quad \tau^{-a} \otimes \tau^{-b} \cong \tau^{-(a+b)} \qquad (\exp_-)$$

This is an isomorphism as it amounts to re-associating products.

In the case that just one coefficient is negative, our notion of inverse satisfies a generalisation of the exponentiation law as a lax property:

$$\forall a, b.\, (a \geq 0 \wedge b \geq 0) \quad \tau^a \otimes \tau^{-b} \multimap \tau^{a-b} \tag{$\exp_\pm$}$$

The lax inverse law $\tau \otimes \tau^{-1} \multimap 1$ is then a specialisation of the above with $a = b = 1$ since $\tau^0 = 1$. As another example, consider $a = 5$ and $b = 3$; then we can eliminate/consume three elements of a quintuple to get pairs $(a - b = 2)$. This raises an interesting combinatorial question about the number of functions inhabiting this type (witnessing this lax property).

▶ **Proposition 1.** *It turns out that the number of possible witnesses of the lax inverse law* $\tau^a \otimes \tau^{-b} \multimap \tau^{a-b}$ *if* $a \geq 0$, $b \geq 0$ *and* $a \geq b$ *is simply* $\binom{a}{b} \times (a-b)! = \frac{a!}{b!}$.

The intuition for this is that if we have some number a of values and some number b of inverses, and we have more values than inverses, then we must apply every single inverse to a value, and the only choice we can make is which values we are going to consume. Combinatorially there are $\binom{a}{b} = \frac{a!}{b!(a-b)!}$ ways to choose b of the a elements, leaving $(a - b)$ elements remaining in the end which we can permute in any order (hence the $(a - b)!$ factor which cancels out). If $b \geq a$, then we must instead consume every value, and so conversely we are simply choosing which a inverses we will use to do this, giving a result of $\frac{b!}{a!}$.

Thus far in the paper we have examined the linear λ-calculus and linear types in Haskell, where inverses have no significant computational content beyond consuming a linear value. However, later in Section 6 we will work in a setting with inverses that incorporate local side effects. Notice that in such a setting the order in which we apply inverses may be important. Thus, we also consider the result we obtain when taking this into consideration.

▶ **Proposition 2.** *If reorderings are considered distinct, then the number of possible witnesses of the lax exponentiation law* $\tau^a \otimes \tau^{-b} \multimap \tau^{a-b}$ *if* $a \geq 0$, $b \geq 0$ *and* $a \geq b$ *is a* factorial *(*$a!$*).*

This is derived by $P(a, b) \times (a - b)!$, i.e. $\dfrac{a!}{(a-b)!} \times (a - b)! = a!$ since again we can "consume" elements via their inverses in any order, giving the number of permutations $P(a, b)$ of length b taken from a, and there are $(a - b)$ remaining elements left afterwards to permute. Here we run through the proof in full, as it is instructive about how to inhabit the law in either case.

Proof. We prove that the number of possible witnesses of the lax exponentiation law is as given above by induction on b (and recall we assume $a \geq b$).

- $(b = 0)$ For the base case, we are then considering $|\tau^a \otimes \tau^0 \multimap \tau^{a-0}| = |\tau^a \multimap \tau^a|$ since $\tau^0 = 1$. The possible inhabitants of $\tau^a \multimap \tau^a$ are then just permutations of the a-wise product of τ and so $|\tau^a \multimap \tau^a| = a!$, giving the result here.
- $(b = 1)$ We next consider another case where $b = 1$ since this is instructive (though not necessary). Here we thus need to find how many ways there are to inhabit $\tau^a \otimes \tau^{-1} \multimap \tau^{a-1}$, i.e., how many ways to "cut out" one τ from an a-wise product of τ.
  We can construct a many terms as witnesses for this type by picking any one of the τ values to consume with the inverse τ^{-1}:

$$\begin{aligned}
&\lambda((a_1, a_2, \ldots, a_n), u) = \mathbf{let}\ () = u\ a_1\ \mathbf{in}\ (a_2, \ldots, a_n) && : (\tau^a \otimes (\tau \multimap 1)) \multimap \tau^{a-1} \\
&\lambda((a_1, a_2, \ldots, a_n), u) = \mathbf{let}\ () = u\ a_2\ \mathbf{in}\ (a_1, a_3, \ldots, a_n) && : (\tau^a \otimes (\tau \multimap 1)) \multimap \tau^{a-1} \\
&\qquad\qquad \ldots \\
&\lambda((a_1, a_2, \ldots, a_n), u) = \mathbf{let}\ () = u\ a_n\ \mathbf{in}\ (a_1, a_2, \ldots, a_{n-1}) && : (\tau^a \otimes (\tau \multimap 1)) \multimap \tau^{a-1}
\end{aligned}$$

Whichever witness we choose, one inverse has been applied, so $a-1$ of the original elements remain. These can be permuted in any order, giving a total of $a \times (a-1)! = a!$ witnesses.

- ($b = n+1$) (Inductive step) We need to show that $|\tau^a \otimes \tau^{-(n+1)} \multimap \tau^{a-(n+1)}| = a!$. We know $\tau^{-(n+1)} \cong \tau^{-1} \otimes \tau^{-n}$ as we can simply "split off" one of the $n+1$ inverses from the remaining product of n inverses. Similarly to the base case ($b = 0$), we first apply this inverse to any of the elements of τ^a; there are a possible choices here. This leaves $a-1$ of the original elements remaining, and n inverses. Therefore we can reason as follows:

$$\begin{aligned}
& |\tau^a \otimes \tau^{-(n+1)} \multimap \tau^{a-(n+1)}| \\
= \; & |(\tau^a \otimes \tau^{-1}) \otimes \tau^{-n} \multimap \tau^{a-(n+1)}| && \text{(split off one inverse, as above)} \\
= \; & a \times |\tau^{a-1} \otimes \tau^{-n} \multimap \tau^{(a-1)-n}| && \text{(a ways to apply one inverse)} \\
= \; & a \times (a-1)! && \text{(induction with } a-1\text{)} \\
= \; & a! && \square
\end{aligned}$$

Note, $a \geq (n+1)$ implies $(a-1) \geq n$, which is needed to inductively apply the proposition. Another way to see this intuitively is as follows. For $|\tau^a \otimes \tau^{-b} \multimap \tau^{a-b}|$ (with $a \geq b$), in order to find a witness we simply need to arrange the a elements in any order, such that the first b are assigned to the b inverses and the remaining $a-b$ are left unused. There are $a!$ ways to arrange the a elements, giving the result. Again, in the case that $b \geq a$ we just consider the b inverses instead, assigning inverses to elements, giving a total of $b!$ orderings. ◀

After that *divertimento* into the inhabitants of the negative exponentiation law ($\exp_\pm$) and its cardinality, we consider the second exponentiation law (sometimes called *power of a product*), which for regular types is the following isomorphism, by commutativity and associativity:

$$\forall a.\, (a \geq 0) \quad \sigma^a \otimes \tau^a \cong (\sigma \otimes \tau)^a \tag{$\exp_\sigma$}$$

For a version of this law with negative exponents, a special case with $a = -1$ is already provided by the monoidal functor structure of Section 3 with $\mathsf{mmult} : \sigma^{-1} \otimes \tau^{-1} \multimap (\sigma \otimes \tau)^{-1}$ which combines inverses. Composing this with the double-negative-exponents law $\mathsf{exp\_} : \tau^{-a} \otimes \tau^{-b} \cong \tau^{-(a+b)}$ yields the generalisation to all negative exponents:

$$\forall a.\, (a \geq 0) \quad \sigma^{-a} \otimes \tau^{-a} \multimap (\sigma \otimes \tau)^{-a} \tag{$\exp_{-\sigma}$}$$

For example, if $a = 2$ then this is derived as:

$$\sigma^{-2} \otimes \tau^{-2} \overset{\alpha+\gamma}{\cong} (\sigma^{-1} \otimes \tau^{-1}) \otimes (\sigma^{-1} \otimes \tau^{-1}) \overset{\mathsf{mmult}\otimes\mathsf{mmult}}{\multimap} (\sigma \otimes \tau)^{-1} \otimes (\sigma \otimes \tau)^{-1} \overset{\mathsf{exp\_}}{\cong} (\sigma \otimes \tau)^{-2}$$

where the isomorphism on the left applies associativity α and commutativity γ.

Overall, negative exponents generalise the "inverses as consumers" story. Given a product of two exponentiated types, one positive and one negative, then "actioning" the inverses involves consuming some number of elements of a product, leaving us with the remainder. This captures the notion of "projection", where some part of a type is thrown away and some part is retained. This pattern is relevant next, where having inverses and negative exponents for regular types allows us to define the derivative of a type *with respect to another type*.

5 Differentiating with Inverses

With our notion of multiplicative inverse in hand, we can apply other ideas from mathematics which rely on the presence of division.

First, let's recall the remarkable feature of regular types (with added type variables) that one can compute their *derivative* by applying the laws of Newton-Leibniz calculus [29]. This produces a companion data type of "one-hole contexts" for the original type, a beautiful idea due to McBride [37]. For example, for the parametric type α^4 (4-tuples with elements all of the same type) its derivative with respect to α is $4\alpha^3$, equivalent to $\alpha^3 + \alpha^3 + \alpha^3 + \alpha^3$. The intuition behind this is that there are four distinct ways in which you can take the original data type (4-tuples) and remove a single element (creating a hole), leaving the surrounding context (a triple of the three remaining elements). We can visualise these four possibilities (each of type α^3) as follows, where $-$ represents the "hole" and where $x, y, z, w : \alpha$:

$$(-, y, z, w) \qquad (x, -, z, w) \qquad (x, y, -, w) \qquad (x, y, z, -)$$

We write this derivative as $\partial_\alpha(\alpha^4)$, i.e., the partial derivative with respect to α keeping other variables constant (including recursion variables or other type parameters).

This approach can be applied to recursive regular types. For example, McBride's technique calculates $\partial_\alpha(\mathsf{List}\ \alpha) = \partial_\alpha(\mu X.1 + \alpha \times X) = (\mathsf{List}\ \alpha) \times (\mathsf{List}\ \alpha)$ representing the idea that a list with a single hole is equivalent to a pair of lists – the prefix of elements before the hole and the suffix of elements after the hole. The data type of list zippers (à la Huet [22]) is then given by $\alpha \times \partial_\alpha(\mathsf{List}\ \alpha)$ where we have a value α which "fills" the hole position, paired with its context. It is possible to extend this notion further to consider derivatives of general data types which act as containers [1] or even data which is in the process of being transformed from one type to another [38], but here we will concentrate on the simpler cases.

A data type of contexts with *two* holes can be obtained by repeated differentiation, i.e., $\partial_\alpha(\partial_\alpha(f(\alpha)))$, and thus we can compute contexts with n holes by taking the n-th derivative. Such holes are all independent; they can appear anywhere in the type. For example, $\partial_\alpha(\partial_\alpha(\mathsf{List}\ \alpha)) = (\mathsf{List}\ \alpha \times (\mathsf{List}\ \alpha \times \mathsf{List}\ \alpha)) + ((\mathsf{List}\ \alpha \times \mathsf{List}\ \alpha) \times \mathsf{List}\ \alpha)$, representing two ways of adding another hole to a list which already has one hole: either we have another hole in the left sublist or in the right sublist.

Though the derivatives above are based on standard regular types which have the structure of intuitionistic logic, linear regular types form a semiring in exactly the same syntactic manner. Thus, the notion of taking the derivative of a type applies equally well in the linear setting. We consider derivatives of linear regular types going forward, and show that inverses allow us to define what it means to take a derivative *with respect to another type*.

First, let's stay on the firm ground of real analysis. Recall from calculus that we can take the derivative of a function f with respect to another function g by the following method:

$$\partial_{g(\alpha)}(f(\alpha)) = \frac{\partial_\alpha(f(\alpha))}{\partial_\alpha(g(\alpha))} \tag{3}$$

For example, taking $f(\alpha) = \alpha^4$ and $g(\alpha) = \alpha^2$, then:

$$\partial_{(\alpha^2)}\alpha^4 = \frac{\partial_\alpha(\alpha^4)}{\partial_\alpha(\alpha^2)} = \frac{4\alpha^3}{2\alpha} = 2\alpha^2 \tag{4}$$

Giving this an interpretation in regular types, rather than $\mathbb{R}$, recall α^4 is a 4-tuple (a quadruple) and α^2 is a 2-tuple (a pair). Differentiating α^4 with respect to α^2 yields $2\alpha^2$ which is the data type capturing the two possible contexts obtained by removing a pair from the original type, leaving two elements in the remaining context. We call the pair removed here a *2-hole*, as it captures two contiguous (adjacent) holes. Note that this is different to the case, described earlier, of differentiating with respect to α twice; this gave two independent holes which did not necessarily have to be contiguous. The resulting type here $2\alpha^2 = \alpha^2 + \alpha^2$ can be interpreted as the type of 2-hole contexts, illustrated as:

$$(-_1, -_2, z, w) \qquad \textit{and} \qquad (x, y, -_1, -_2) \tag{5}$$

where $-_1$ and $-_2$ correspond to the two successive components of the 2-hole. We are not allowed to have the 2-hole splitting the remaining two elements like $(x, -_1, -_2, y)$; the data type of 2-hole contexts views the whole type in terms of pairs.

The derivative calculated in (4) was in $\mathbb{R}$ and *then* we interpreted the result as a type. This however does not generalise well. Consider the derivative of α^3 with respect to α^2:

$$\partial_{(\alpha^2)}\alpha^3 = \frac{\partial_\alpha(\alpha^3)}{\partial_\alpha(\alpha^2)} = \frac{3\alpha^2}{2\alpha} = \frac{3}{2}\alpha$$

We simplified the result following the axioms of fields here but are left with the unwieldy $\frac{3}{2}$ which we cannot meaningfully translate into the realm of types. Using our approach of multiplicative inverses instead yields a more generally applicable result for regular types.

▶ **Definition 2.** The derivative of a regular type with respect to another regular type is:

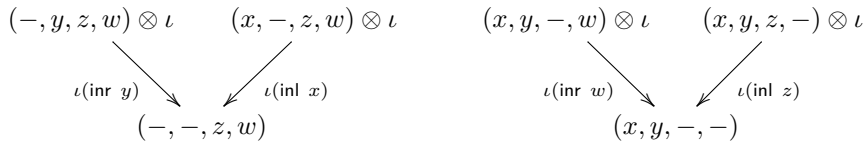
$$\partial_{g(\alpha)} f(\alpha) = \partial_\alpha f(\alpha) \otimes (\partial_\alpha g(\alpha))^{-1} \tag{6}$$

This construction yields the usual derivative of the numerator, paired with a consumer of the derivative of the denominator.

Returning to the example of taking the derivative of α^4 with respect to α^2 shown in equation (4), we instead apply the inverses approach of Definition 2 which yields:

$$\partial_{\alpha^2}(\alpha^4) = 4\alpha^3 \otimes (2\alpha)^{-1}$$

This is the type $4\alpha^3$ of 1-hole contexts for 4-tuples (the four possible triples resulting from removing one element) paired with an inverse which can be used to consume a further α value to create a 2-hole. This inverse consumes 2α values, i.e., $\alpha \oplus \alpha$, where the α value is tagged with an extra bit of information to explain to the inverse which element is being removed to create the 2-hole. We can see this as the four possible 1-holes with an inverse $\iota : (2\alpha)^{-1}$ which can be actioned to recover the 2-holes for 4-tuples as in equation (5):



(7)

with the usual injections $\mathsf{inl} : A \multimap A \oplus B$ and $\mathsf{inr} : B \multimap A \oplus B$.

We put all this together in Haskell to define a notion of 1-hole contexts for 4-tuples (`QuadContexts` below), which we pair with consumers to create 2-holes (`QuadTwoContexts` below):

```
-- Represents 4a^3 (the four possible ways to remove one element from a 4-tuple).
data QuadContexts a =
    Mk1 a a a -- context: -, y, z, w
  | Mk2 a a a -- context: x, -, z, w
  | Mk3 a a a -- context: x, y, -, w
  | Mk4 a a a -- context: x, y, z, -

data QuadTwoContexts a = Mk (QuadContexts a) (Inverse (Either a a))
```

We can then use the inverses, as in the above illustration, to map from a 2-hole and a context back into the original α^4 type, implementing the illustration of equation (7):

```
fromContext :: (a, a) %1 -> QuadTwoContexts a %1 -> (a, a, a, a)
-- In the first two cases, we put the 2-hole at the start of the 4-tuple.
fromContext (h1, h2) (Mk (Mk1 y z w) inv) =
    letUnit (inv (Right y)) (h1, h2, z, w)  -- consume y then fill 2-hole

fromContext (h1, h2) (Mk (Mk2 x z w) inv) =
    letUnit (inv (Left x))  (h1, h2, z, w)  -- consume x then fill 2-hole

-- In the second two cases, we put the 2-hole at the end of the 4-tuple.
fromContext (h1, h2) (Mk (Mk3 x y w) inv) =
    letUnit (inv (Right w)) (x, y, h1, h2)  -- consume w then fill 2-hole

fromContext (h1, h2) (Mk (Mk4 x y z) inv) =
    letUnit (inv (Left z))  (x, y, h1, h2)  -- consume z then fill 2-hole
```

The intuition is that the inverse $(2\alpha)^{-1}$ (`inv` above) is used to consume an element of the 4-tuple that overlaps with the hole, with the constructor of `Either` delineating from which position we are consuming.

This technique becomes more useful when we want 2-hole contexts in a type which does not contain an even number of elements–or more generally when we want n-hole contexts from a data type whose number of elements are not exactly divisible by n. For example, we can now compute the type of 5-tuples with 2-holes as:

$$\partial_{(\alpha^2)}\alpha^5 = 5\alpha^4 \otimes (2\alpha \multimap 1)$$

The usual interpretation in the real domain would have yielded $\frac{5}{2}\alpha^3$ for which we have no interpretation in regular types. Instead, we can use the inverses approach to yield contexts of 2-holes for 5-tuples. The resulting equivalent of `fromContext` then has to capture a final hole which overlaps the preceding one, to deal with the fact that 5 is not factored by 2.

An even more interesting possibility presents itself, however. Note that the above example of 2-hole contexts for 4-tuples considers the context to also be chunked into contiguous pairs, and thus we cannot have the context $(x, -_1, -_2, w)$ with the 2-hole "in the middle". However, such an interpretation should certainly be possible using the inverse approach, as there is enough information available: in the domain of $\mathbb{R}$, division (multiplying with an inverse) is a non-injective operation (it destroys information) whereas with regular types, the inverse preserves the structure of the original type until we apply `divide`. Indeed, we can define the following alternate way of mapping the `QuadTwoContexts` data type back to a 4-tuple:

```
fromContext' :: (a, a) %1 -> QuadTwoContexts a %1 -> (a, a, a, a)
fromContext' (h1, h2) (Mk (Mk1 y z w) inv) =
    letUnit (inv (Left h1)) (h2, y, z, w)  -- first hole outside the 4-tuple!

fromContext' (h1, h2) (Mk (Mk2 x z w) inv) =
    letUnit (inv (Left x)) (h1, h2, z, w)  -- 2-hole at start of the 4-tuple

fromContext' (h1, h2) (Mk (Mk3 x y w) inv) =
    letUnit (inv (Left y)) (x, h1, h2, w)  -- 2-hole in middle of the 4-tuple

fromContext' (h1, h2) (Mk (Mk4 x y z) inv) =
    letUnit (inv (Left z)) (x, y, h1, h2)  -- 2-hole at end of the 4-tuple
```

Compared to `fromContext`, this "shifts" the 2-hole through successive positions of the context, not requiring that the surrounding context is broken up into pairs. Instead, `fromContext'` uses the inverse to consume the extra value always under the left component of the hole, leaving the remaining four elements as follows:

$$\cancel{h_1}(h_2, y, z, w) \quad (\cancel{x}\ h_1, h_2, z, w) \quad (x, \cancel{y}\ h_1, h_2, w) \quad (x, y, \cancel{z}\ h_1, h_2)$$

with the consumed element shown here in strikethrough. Note in particular the first case (`Mk1`) where the deleted element is "outside" the 4-tuple; if we have a 2-hole where the second hole is at the leftmost position of a 4-tuple, then the first hole refers to data (h_1) outside the 4-tuple to the left. We can understand such data as being a "boundary value", which hints at an application for n-hole context data types: *stencil computations*.

n-holes and stencil computations

Typically applied to arrays, a stencil computation traverses each position in an array, reading a small neighbourhood of elements at and around the "current" position to compute the corresponding element of a new array. This is used e.g., for image processing (e.g., Gaussian blur), cellular automata, and the finite-difference method for solving PDEs [25]. The idea can be generalised to types other than arrays, e.g., trees, graphs, and triangular meshes [41]. The above notion of a generalised derivative with respect to another type and its interpretation as a kind of zipper on n-holes captures exactly this structure. In the above example of $\alpha^2 \otimes \partial_{\alpha^2}(\alpha^4)$, the 2-hole describes a neighbourhood of two elements, looking in this case "to the left" (in other words, the neighbourhood comprises the current element and one to its left in the above interpretation). In the `Mk1` case above, the value h_1 is the boundary value when the second hole is positioned at the leftmost point. This is standard for stencil computations: we need a "halo" of boundary values to compute at a data structure's edge.

As a more concrete example, consider the discrete Laplace operator over 1-dimensional arrays. Mathematically, we can describe the operation as taking an array A and computing the elements at position i in the output array B as follows:

$$B_i = A_{i-1} - 2A_i + A_{i+1}$$

(Note this ignores what to do at the boundaries of the array at positions A_{-1} and A_{n+1}). This can be structured using a local computation over 3-holes, e.g., in Haskell:

```
laplace :: (Float, Float, Float) %1 -> Float
laplace (a, b, c) = a - 2*b + c
```

We can then capture the data type of contexts for the corresponding global traversal of a data structure (say lists) by computing the 3-hole contexts, e.g.,

$$\partial_{\alpha^3}(\mathsf{List}\ \alpha) = \partial_\alpha(\mathsf{List}\ \alpha) \otimes (3\alpha^2 \multimap 1)$$

which gives us the usual data type of 1-hole contexts plus a way to consume 2 elements, yielding a gap for 3-holes. A recursive function can then navigate "right" through this data structure, applying `laplace` to every 3-hole to compute the values of an output list, giving one iteration of the discrete Laplace stencil computation. The actual definition of this operation is less relevant to type inverses so we elide it here, but include it in the accompanying code artifact for this pearl. The inverse is then needed to map this zipper data structure back to the original list form, via an operation akin to `fromContext'`.

Thus, our notion of multiplicative inverse has given us a way to generalise derivatives to n-holes, which can then be used to capture the "sliding window" of a stencil computation through any data type.

6 Communicating with Inverses

So far the inhabitants of inverses τ^{-1} have been rather mundane, consuming their inputs by pattern matching. We now turn to a richer setting in which some types have an inverse inhabited by functions which can perform some kind of local side effect. For this we use the functional language Granule which combines linear and indexed types with graded modal types [42], though we will mostly leverage just linear types here. We consider richer inverses first through the question of whether our notion of inverses is an *involution*.

A function is an involution if it is its own inverse, i.e., $f(f(x)) = x$. In abstract algebra, the inverses in groups and fields are automatically involutions.[5] In the setting of Granule, the inverse of a type is also an involution with isomorphism $(\tau^{-1})^{-1} \cong \tau$, which might be surprising given that so far our inverse has been lax.

One direction of the involution isomorphism $\tau \multimap (\tau^{-1})^{-1}$ is easy via function application. We give the definition below in Granule, whose syntax closely resembles Haskell's:

```
type Inverse a = a → ()

-- i.e. the type expands to a → ((a → ()) → ())
invol : ∀ {a : Type} . a → Inverse (Inverse a)
invol x = λf → f x
```

As one can see, the differences between Granule and Haskell are fairly minimal: Granule's arrows are linear by default (fans of lollipops $\multimap$ will just have to squint in code samples!) whereas in Haskell's linear types extension the linear multiplicity must be explicitly written. The remainder of the translation from Haskell to Granule mainly lies in explicitly quantifying our types and using `:` rather than Haskell's `::` for our type declarations. If the reader would like to follow along using Granule for this section, we recommend the latest release.[6]

Usefully for this pearl, Granule implements a mechanism for algorithmically deriving a weakening operation for regular types [23]; this can be accessed by writing `drop @t` for some type `t`, so we have for example `drop @Bool : Bool → ()`, providing an inverse.

This direction of the involution isomorphism is of course also well-defined in the linear λ-calculus and in Haskell. The opposite direction $(\tau^{-1})^{-1} \multimap \tau$ is much more challenging: in fact it is not inhabited if we restrict ourselves to the linear λ-calculus or even traditional non-linear Haskell, but it is instead a *sequentially-realizable function* [32, 33].

A sequentially-realizable function is one which has outwardly pure behaviour but relies on a notion of local side effects; these are entirely contained within the body of the function. Traditionally, sequentially-realizable functions have only been expressible in the ML-family of languages (which allow unrestricted side effects) but not at all in Haskell. However, in the context of linear typing, we can now safely re-introduce some side effects via linear references or linear channels. We show the latter approach, leveraging Granule's session-typed linear channels [35] inspired by the GV calculus [54]. The same approach would work equally well in other implementations of similar calculi, such as Fst [31] or FreeST 2 [3].

Originating from Gay and Vasconcelos [17], further developed by Wadler [54], for which we use the formulation of Lindley and Morris [30], the GV calculus extends the linear λ-calculus with a type of channels parameterised by session types [57], which capture the protocol of interaction allowed over the channel. Granule provides an analogous type of linear channels `LChan : Protocol → Type` indexed by protocols, given by the constructors:

---

[5] For a group $(X, \bullet, e)$ then the properties of inverses yield $(x^{-1})^{-1} \bullet x^{-1} = e$ which implies $(x^{-1})^{-1} \bullet x^{-1} \bullet x = x$; thus, by the same properties we have $(x^{-1})^{-1} = x$.

[6] Examples were tested on `https://github.com/granule-project/granule/releases/tag/v0.8.1.0`

```
Send : Type → Protocol → Protocol          End : Protocol
Recv : Type → Protocol → Protocol
```

The following primitives are then provided for using these (synchronous) channels:

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

This is a subset of the available primitives. We can see that `send` takes an input channel with protocol `Send a s` and an input value `a` which is sent over the channel to yield a channel which can be used according to protocol `s`. The `recv` function is dual, taking a channel which is allowed to follow protocol `Recv a s`, returning a pair of the received `a` value and a new channel that can behave as `s`. The `close` primitive consumes a channel which is at the end of a protocol. Lastly, `forkLinear` spawns a process from the parameter function, which is applied to a freshly created channel, returning a channel with the "dual" protocol in order to communicate with the new process. Here, `Dual` is a type-level function defined:

```
Dual (Send a s) = Recv a (Dual s)          Dual End = End
Dual (Recv a s) = Send a (Dual s)
```

Interestingly, `forkLinear` is a combinator relating inverse and duality: $(\texttt{LChan s})^{-1} \multimap \texttt{LChan}\ (\texttt{s}^{\perp})$, that is, a function consuming a channel with behaviour `s` yields a channel with dual behaviour (denoted by the standard notation $\perp$ as in linear logic and the GV calculus).

We now have adequate machinery to define involution in the direction $(\tau^{-1})^{-1} \multimap \tau$:

```
1  involOp : ∀ {a : Type} . Inverse (Inverse a) → a
2  involOp k =
3    let r       = forkLinear (λs → k (λx → let c = send s x in close c));
4        (x, c') = recv r;
5        ()      = close c'
6    in x
```

Thus `k` has type `(a → ()) → ()`, to which the function `λx → let c = send s x in close c` is passed; this sends the input `x : a` on the channel `c` which is then closed. This channel is provided by `forkLinear`, and so `k` is applied in a process taking one end of the channel to "sneak out" the value of type `a`. Outside this, `recv` waits to receive from the dual end of the channel returned by `forkLinear`. The remaining channel is closed and `x` is returned. A local side effect is performed within `involOp` which is not observable externally, but it would not have been possible to construct the required function without carrying out this effect.

In languages such as ML, an alternate equivalent definition can be given using mutable references which also resembles Longley's **F** combinator [32].

Lastly, the functions `invol` and `involOp` form an isomorphism, witnessing $\tau \cong (\tau^{-1})^{-1}$. The proof of this can be shown via calculating on the definitions, with details given in Appendix B. We refer the reader to Lindley et al. [30, 31] for details of how adding session-typed linear channels to the linear λ-calculus (and linear System F) retains type safety.

Session types based on linear channels can also be represented in Haskell, but they do not allow us to demonstrate a full isomorphism to the same extent. Here we illustrate the more challenging direction of the involution using the Priority Sesh library,[7] a recent package for

[7] Available at `https://github.com/wenkokke/priority-sesh`

session-typed communication in Linear Haskell which is itself inspired by the GV calculus [27]. However, note that the two directions cannot quite form an involution here, since everything must be wrapped up in the linear IO monad for these session types to be used; we cannot confine the side effects to the function as is possible in Granule.

```
type Inverse' a = a %1 -> Linear.IO ()

involOp :: Inverse' (Inverse' a) %1 -> Linear.IO a
involOp k = do
  (s, r) <- new
  void $ forkIO $ k (\z -> send s z)
  recv r
```

Continuation monad

A double inverse type $(\tau^{-1})^{-1} = (\tau \multimap 1) \multimap 1$ is also a specialisation of the familiar *continuation monad* [52], whose return type is the unit type (the Haskell data type is often written `data Cont r a = Cont ((a -> r) -> r)` so this is `Cont () a` here). The definition of `invol` provides the `return` operation of the monad and the "bind" operator is the usual definition for the continuation monad:

```
return : ∀ {a : Type} . a → Inverse (Inverse a)
return = invol

bind : ∀ {a b : Type}
     . Inverse (Inverse a) → (a → Inverse (Inverse b)) → Inverse (Inverse b)
bind m k = λc → m (λa → k a c)
```

A standard way of understanding the use of the continuation monad is to see that its Kleisli arrows (functions of type `a → Inverse (Inverse b)`) characterise continuation-passing style (CPS) programs which can then be sequentially composed. This can be seen via a little algebraic manipulation:

$$\texttt{a} \to \texttt{Inverse (Inverse b)} \;\equiv\; \texttt{a} \to \texttt{((b} \to \texttt{())} \to \texttt{())} \;\cong\; \texttt{(b} \to \texttt{())} \to \texttt{(a} \to \texttt{())}$$

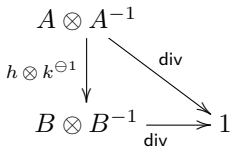
Thus we can see that the Kleisli arrows are CPS-transformed functions of "$a \to b$". Under our interpretation, these are the same as functions that map consumers of b to consumers of a, and the "double inverse" monad gives us a sequential composition for these inverses.

The usual way to "evaluate" a continuation monad computation is to end up with a value of type `Cont r r` (i.e., `(r -> r) -> r`) to which the identity is applied to return the "final" value of type `r`. This requires that the `r` type of the whole `Cont r` computation is pre-determined based on what value we want to be able to extract from a continuation monad computation. For `Inverse (Inverse a)` we can only apply the identity when `a = ()`, i.e., only in trivial cases. However, our "double inverse" monad is actually more powerful thanks to linearity and sequential realizability: we can extract the value of any `Inverse (Inverse a)` computation for any `a` by applying the sequentially-realizable involution function `involOp` to extract the `a` value. This ends up being more flexible than the continuation monad since we need not pre-determine the continuation "result" type (which for the double inverse is fixed as unit anyway) and we can extract the value inside the computation at any point.

Thus, viewing the continuation monad through the lens of linear-logical inverses yields a more flexible continuation-passing style monadic composition. The crucial restriction, though, is that the continuations must be used *linearly*.

Calculating with inverses that communicate

Recall the naturality property discussed in Section 2:



In Granule with local side effects due to channels, this equation only holds if $k \circ h = id$. Consider the following code where `divNat` captures the left-bottom path of the diagram:

```
divNat : ∀ {a b : Type} . (a → b) → (b → a) → (a, Inverse a) → ()
divNat h k (x, y) = divide (h x) (comap k y)

example : ∀ {a b : Type} . (a → b) → (b → a) → a → a
example h k a =
  let r = forkLinear (λs → divNat h k (a, λy → let c = send s y in close c));
      (a', c') = recv r;
      () = close c'
  in a'
```

The `example` function applies `divNat` inside a forked process where the inverse sends the result on the channel which is received on the outside. `example h k` is only the identity function if `k . h` $= id$, e.g., `example (λx→ x + 1) (λx→ x - 1) 42` evaluates to `42`. Thus, we can see the power of the local side effects; here inverses can do more than just consume.

7 Additive Inverses

As we have demonstrated, a reasonable definition of inverses exists for product types in the realm of linear logic. One might therefore wonder whether defining inverses for sum types (i.e., *subtraction*) is also feasible. In much the way that defining a multiplicative inverse almost gives us a semifield of types because some of the identities are lax, being able to define an additive inverse would similarly give us something closely approximating a ring of types. The answer as to whether we can do this, however, is somewhat mixed.

The linear regular types we have been using have product types as linear logic's "multiplicative conjunction" $\otimes$ and sum types as "additive disjunction" $\oplus$. Unfortunately, we cannot define a sensible additive inverse for this operator. To do so we would need to have $A \oplus -A \cong 0$. However, defining a map $A \oplus -A \multimap 0$ is impossible regardless of the value of $-A$, because if A is nonempty then $A \oplus B$ must also be nonempty which means we cannot construct a term of type 0. Furthermore, defining a map $0 \multimap A \oplus -A$ is impossible unless $A = -A = 0$, as otherwise we would have to be able to construct some value of either type A or type $-A$ from nothing. Consequently, linear regular types cannot form a *field* (with both multiplicative and additive inverses).

It turns out that the reason it is not possible to define an additive inverse in the context of standard intuitionistic logic or while using the $\oplus$ operation offered by linear regular types is a corollary to a result known as *Crolard's lemma* [5]. This lemma states that subtraction $A \setminus B$ cannot be defined for disjoint unions in the category of sets unless either A or B is the empty set. In fact, this result also applies to any operation like $\oplus$ which allows a free choice between A and B, so any definition of subtraction with respect to $\oplus$ must be trivial.

However, defining products as multiplicative conjunction and sums as additive disjunction as in regular types is not the only possible interpretation we can use for these concepts. We can just as easily have product types that follow the rules of the & operator, pronounced "with" and describing "additive conjunction", and similarly we can have sum types based on the ⅋ operator, pronounced "par", which describes "multiplicative disjunction". This makes it possible to discuss a closely related setting which we will call *coregular types* (as these operations behave in a dual manner to those used to define *regular types*), with type syntax:

$$\tau ::= \tau \,\&\, \tau' \mid \tau \parr \tau' \mid \top \mid \bot$$

where $\top$ and $\bot$ are the units for & and ⅋ respectively. The intuition for & is that $a \,\&\, b$ allows us to select one of a or b and use it, rather than having access to both at the same time but having to use each one, as with $a \otimes b$. The ⅋ operator is more difficult to understand intuitively, but one interpretation is that $a \parr b$ gives two processes a and b that happen in parallel, and we have the choice of how to interleave the two processes [4]. The important thing to keep in mind is that & is dual to $\oplus$ and ⅋ is dual to $\otimes$.[8]

If we consider the coregular ⅋ sum types which do not allow a free choice between A and B rather than the regular $\oplus$ sum types, then the outlook for defining an additive inverse is less bleak: it is possible to define an inverse operation to multiplicative disjunction. Cointuitionistic linear logic [5] offers an operation called *linear subtraction*, denoted $A \setminus B$ and read "A *excludes* B", which acts as the left adjoint of ⅋. Intuitively, we can understand linear implication in the following way:

$$A \otimes B \vdash C \text{ if and only if } A \vdash B \multimap C$$

which arises from the categorical notion of adjunctions: $(- \otimes B)$ is left adjoint to $(B \multimap -)$. Dually, linear subtraction can be understood as follows:

$$A \vdash B \parr C \text{ if and only if } A \setminus B \vdash C$$

In other words, if A gives us $B \parr C$ then A *excluding* the possibility of B gives us C, and conversely if A excluding B gives C then from A we can get $B \parr C$.

In this dual setting, we must now find a definition of subtraction from a suitable unit which acts as the additive inverse we desire. Since linear subtraction is dual to linear implication, just as we can define implication in terms of the ⅋ connective (i.e. $A \multimap B \equiv A^\perp \parr B$), we can similarly represent subtraction using the $\otimes$ connective, as $B \setminus A \equiv B \otimes A^\perp$.

Using this representation of linear subtraction, by duality we can show that a nontrivial inverse to multiplicative disjunction ⅋ exists in the context of linear type theory. Dually to our definition of an inverse for multiplicative conjunction, we can define an inverse to multiplicative disjunction as $-\tau \triangleq \bot \setminus \tau$, via linear subtraction as discussed above.

Similarly to the lax identity $\tau^{-1} \otimes \tau \multimap 1$, the additive inverse $-\tau$ also satisfies a lax inverse law, but in the opposite direction:

$$\bot \multimap \tau \parr -\tau \tag{8}$$

Via the identity between $\multimap$ and ⅋ and between $\setminus$ and $\otimes$ then $\tau \parr -\tau \cong \tau^\perp \multimap (\bot \otimes \tau^\perp)$. If we had a type system involving both regular and coregular types with a duality operator the above lax law (8) would be given constructively by the term $\lambda b.(\lambda x.(b, x)) : \bot \multimap \tau^\perp \multimap (\bot \otimes \tau^\perp)$.

[8] Classical linear logic has an involutive duality operator, written $(-)^\perp$, where $(A \,\&\, B)^\perp = A^\perp \oplus B^\perp$ and $(A \parr B)^\perp = A^\perp \otimes B^\perp$.

The applications of this identity are less apparent, as we cannot construct a witness for it in the same way as the lax inverse law from Section 2 given the constraints of having to choose between working with either regular or coregular types. If we were not constrained by this limitation, we could have an algebraic structure with all four common mathematical operations, with $\bot$ acting as an additive identity and 1 acting as a multiplicative identity.[9] Intuitionistic and cointuitionistic logic can be combined into a single framework, known as bi-intuitionistic logic, and work on making sense of this through the lenses of type theory and category theory is ongoing [15]; this could provide a way to combine regular and coregular types in a single system.

Given the above definitions we can show a lax involution in one direction for additive inverses. Between multiplicative conjunction and disjunction there is a distribution: $(A \otimes (B \parr C)) \multimap ((A \otimes B) \parr C)$ which is not an isomorphism, but it is a valid implication in this direction [13]. Using this weak distribution, for all τ this lax involution is defined:

$$\begin{array}{rcl} -(-\tau) & \cong & (\bot \setminus (\bot \setminus \tau)) \\ & \cong & \bot \otimes (\bot \otimes \tau^{\bot})^{\bot} \\ & \cong & \bot \otimes (1 \parr \tau) \\ & \multimap & (\bot \otimes 1) \parr \tau \\ & \cong & \bot \parr \tau \\ & \cong & \tau \end{array}$$

Interestingly, this kind of dichotomy between additive and multiplicative disjunction can also be seen for conjunction. The multiplicative inverse we have defined for linear regular types does not behave well if we attempt to apply it to the & operator (additive conjunction). We *cannot* define a map $A \,\&\, (A \multimap 1) \multimap 1$, because we can only use one of the two components of the & on the left so we cannot apply the inverse to the A value. Furthermore, similarly to $\otimes$, we cannot in general define a map in the other direction as we would need to be able to construct a value of an arbitrary type A from nothing.

In the end, linear logic cannot yet give us a field of types – it can only afford to give us a ring or half a field (a semifield), but both at once is beyond our current budget. The semifield interpretation however has the closest intuitions to familiar concepts in functional programming, and linear regular types are certainly more frequently encountered than coregular ones in the current programming landscape, hence our focus on them in this pearl.

8 Discussion: Thinking with Inverses

As we near the end of our journey, we remark on some alternate perspectives and approaches, and some connections with related work.

Curry-Howard with inverses

From a logical standpoint, the (lax) inverse property we have discussed provides a natural notion of inverse elimination and introduction in a natural deduction logic for a Curry-Howard correspondent to a type's inverse:

[9] We still cannot, however, form a field even if we use $\otimes$ and $\parr$ as our operations; they do not obey distributivity, $A \otimes \bot \not\cong \bot$ (note for example that $\top \otimes \bot \cong \top$), and indeed both types of inverse only obey lax inverse laws, so the required isomorphisms for a field do not exist.

$$\frac{\Gamma \vdash p \qquad \Gamma \vdash p^{-1}}{\Gamma, \Gamma' \vdash 1}\,{}^{-1}E \qquad \frac{\Gamma, p \vdash 1}{\Gamma \vdash p^{-1}}\,{}^{-1}I$$

i.e., elimination is just a specialised modus ponens and an inverse p^{-1} is introduced by a (linear) proof starting with p and concluding 1 – the subproof consumes the assumption p.

Duality

One may consider trying to use the classical linear logic notion of "duality" $\tau^\perp$ [20] to provide multiplicative inverses, but it does not behave accordingly: $1 \otimes 1^\perp = 1 \otimes \perp = \perp$ but instead we would like $1 \otimes 1^{-1} \cong 1$. However, there are various interesting applications of linear and classical duality relating call-by-value and call-by-name [53, 45]. These take advantage of various properties of duality, some of which our inverses do indeed share.

Negative and fractional types

Despite the algebraic manipulation of data types producing a rich source of ideas, inverses appear to have not had much consideration. One notable thread though is due to James and Sabry, who consider *negative* and *fractional* types in the context of reversible computations where a reciprocal $1/b$ "*imposes constraints on [its] context*" acting as a logical variable [24]. They present a reversible calculus admitting isomorphisms $\eta : 1 \leftrightarrow (1/b) \times b$ for all types: with left-to-right direction producing a fresh logical variable α inhabiting b along with its dual, and the inverse η^{-1} corresponding to unification of logical variables. Later they interpret this categorical semantics computationally [12], defining a sound operational semantics for such types, in which a negative type represents a computational effect that "reverses execution flow" and a fractional type represents one that "garbage collects" values or throws exceptions. This differs from our approach but certainly has some of the same flavour. We cannot however construct a pair of a $b^{-1} \otimes b$ out of thin air for any b.

Cardinalities

As recalled in the introduction, the cardinality operation on types is a semiring homomorphism from regular types to natural numbers (e.g., $|a \times b| = |a||b|$). So what is the cardinality of an inverse type? In a Cartesian setting a function $\tau \to 1$ simply has cardinality 1 since $|\tau \to 1| = |1|^{|\tau|} = 1$. In a linear setting without side effects (i.e., linear channels), we can recover a similar result. As a simple example, consider the boolean type. The cardinality of `Bool` is 2 as it has two elements: `True` and `False`. The cardinality of `Inverse Bool`, on the other hand, is 1; the type has exactly one inhabitant: the `boolDrop` function shown in Section 2.

We could however consider a different notion of cardinality that would allow for $|\tau^a| \times |\tau^b| = |\tau^{a+b}|$ in general; this statement already holds for $a \geq 0 \wedge b \geq 0$, but now we consider cases where the types are not necessarily isomorphic. In particular, we can examine the notion of *fractional cardinalities* [44, 46] assigning a generalised cardinality of $(1/|\tau|)^m$ to τ^{-m}.

If we specialise this we can let $a = 1$ and $b = -1$ and see that $|\tau \otimes \tau^{-1}| = |\tau| \times |\tau^{-1}| = n \times (1/n) = 1 = |1|$; the two types have the same fractional cardinality even though we only have a lax map from $\tau \otimes \tau^{-1}$ to 1. Of course, this does not match up with the standard idea of cardinality on types, as it is clear that $\tau \otimes \tau^{-1}$ has at least as many inhabitants as τ. We leave an interpretation for this as something for others to ponder.

Taylor series

As we have seen, it is not possible to form a field out of regular types (linear or otherwise), because the additive operation that permits an inverse is not the same addition which behaves like the logical $\oplus$ we would usually want. But it turns out that if we suspend our disbelief and assume that types do form a field, some results from real analysis can be applied with surprising success: Taylor series approximations can yield solutions to recursive types. Consider the recursive definition of lists over elements of type α:

$$\mathsf{List}\ \alpha = 1 \oplus (\alpha \otimes \mathsf{List}\ \alpha)$$

Through some unjustified algebraic rearrangement we get $\mathsf{List}\ \alpha = \frac{1}{(1-\alpha)}$ on which we can compute the Taylor expansion yielding the familiar least fixed-point solution of $\mathsf{List}\ \alpha$:

$$\mathsf{List}\ \alpha = 1 \oplus \alpha \oplus \alpha^2 \oplus \alpha^3 \oplus \ldots$$

i.e. a list is either empty, or has one element, or has two elements, etc.

This is quite surprising; we must apply the equations of a field to yield a derivation for this result, using inverses we do not have access to in the realm of regular types, and yet we end up with a result that makes sense using only regular operations. Whether there is an interpretation of our inverses that can lend a meaningful foundation to these intermediate manipulations is unclear, but would certainly be interesting to look into.

One might wonder whether it is coincidental that this result happens to hold true for lists in particular, but this is not the case. Fiore and Leinster [16] show that, for all complex numbers t and polynomials p, q_1 and q_2 with non-negative coefficients (with some restrictions), then if $t = p(t)$ implies $q_1(t) = q_2(t)$ the same result also holds up to isomorphism in any other semiring (as well as for complex numbers), which includes regular types.

This was applied to great effect for the example of finite binary trees to demonstrate the famous "seven trees in one" result [9], showing that there is a particularly elementary bijection (involving case distinctions only down to a fixed depth) between the set T of finite binary trees and the set T^7 of 7-tuples of such trees. It is more difficult to find solutions to more complex types via this kind of equational reasoning, though, particularly due to the Abel-Ruffini theorem [2] which states that there is no solution in radicals to general polynomial equations of degree five or higher.

9 Epilogue

Summary

Taking $\tau^{-1} \triangleq \tau \multimap 1$ yields a useful notion of multiplicative inverse for linear regular types. We have seen this yields (lax) exponentiation laws in the presence of negative coefficients:

$$\tau \otimes \tau^{-1} \multimap 1 \qquad \tau^a \otimes \tau^{-b} \multimap \tau^{a-b} \qquad \tau^{-a} \otimes \tau^{-b} \cong \tau^{-(a+b)}$$
$$\sigma^{-a} \otimes \tau^{-a} \multimap (\sigma \otimes \tau)^{-a} \qquad \tau \multimap (\tau^{-1})^{-1}$$

for all $a \geq 0, b \geq 0$. The first lax identity is generalised by the second (Section 4). The fourth is induced by $-^{-1}$ being a monoidal functor (Section 3). The last lax identity becomes an isomorphism $\tau \cong (\tau^{-1})^{-1}$ when sequentially realizable functions are permitted (Section 6).

Fin

The algebraic characteristics of data types have been studied and leveraged since the dawn of functional programming; we call them "algebraic" data types, after all. In the linear setting, the idea of consumption as a lax multiplicative inverse has given us a fresh perspective on the algebraic characterisation of regular linear types. Now that linear typing is becoming more mainstream, e.g., in Haskell [6], and with closely related ideas arising in languages like Clean and Rust (the concept of uniqueness which is in some sense dual to linearity [21, 14, 36], and more sophisticated systems tracking ownership and borrowing [56]), this is now an ideal time to start taking our algebraic understanding of data types to the next level. This pearl has been a demonstration of how one weird trick can lead to a journey through many interesting and diverse areas of our field. We hope that that this has stoked your enthusiasm for investigating the idea of taking the inverse of a type even further.

References

1 Michael Abbott, Thorsten Altenkirch, Conor Mcbride, and Neil Ghani. ∂ for data: Differentiating data structures. *Fundam. Inf.*, 65(1–2):1–28, January 2005.

2 Niels Henrik Abel. Mémoire sur les equations algébriques, où l'on démontre l'impossibilité de la résolution de l'équation générale du cinquième degré. 1:28–33, 1824. doi:10.1017/CBO9781139245807.004.

3 Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. Polymorphic context-free session types, 2021. arXiv:2106.06658.

4 Federico Aschieri and Francesco A. Genco. Par means parallel: Multiplicative linear logic proofs as concurrent functional programs. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371086.

5 Gianluigi Bellin, Massimiliano Carrara, Daniele Chiffi, and Alessandro Menti. Pragmatic and dialogic interpretations of bi-intuitionism. Part I. *Logic and Logical Philosophy*, 23(4):449–480, 2014.

6 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.

7 Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice-Hall, Inc., USA, 1997.

8 Richard S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

9 Andreas Blass. Seven trees in one. *Journal of Pure and Applied Algebra*, 103(1):1–21, 1995.

10 Edwin Brady. Idris 2: Quantitative Type Theory in Practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2021.9.

11 Jacques Carette and Amr Sabry. Computing with semirings and weak rig groupoids. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, pages 123–148, Berlin, Heidelberg, 2016. Springer-Verlag.

12 Chao-Hong Chen and Amr Sabry. A computational interpretation of compact closed categories: Reversible programming with negative and fractional types. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434290.

13 J.R.B. Cockett and R.A.G. Seely. Weakly distributive categories. *Journal of Pure and Applied Algebra*, 114(2):133–173, 1997. doi:10.1016/0022-4049(95)00160-3.

14 Edsko de Vries, Rinus Plasmeijer, and David M Abrahamson. Uniqueness typing simplified. In *Symposium on Implementation and Application of Functional Languages*, pages 201–218. Springer, 2007. doi:10.1007/978-3-540-85373-2_12.

15 Harley Eades III and Gianluigi Bellin. A cointuitionistic adjoint logic, 2017. arXiv:1708.05896.

16 Marcelo Fiore and Tom Leinster. Objects of categories as complex numbers. *Advances in Mathematics*, 190(2):264–277, 2005. doi:10.1016/j.aim.2004.01.002.

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

18 Jeremy Gibbons. Calculating functional programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 151–203. Springer, 2002.

19 Jeremy Gibbons. The school of Squiggol - A history of the Bird-Meertens formalism. In *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part II*, pages 35–53, 2019. doi:10.1007/978-3-030-54997-8_2.

20 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

21 Dana Harrington. Uniqueness logic. *Theoretical Computer Science*, 354(1):24–41, 2006.

22 Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

23 Jack Hughes, Michael Vollmer, and Dominic Orchard. Deriving distributive laws for graded linear types. In *TLLA/Linearity*, 2020.

24 Roshan P James and Amr Sabry. The Two Dualities of Computation: Negative and Fractional Types. Technical report, Indiana University, 2012.

25 Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 51–60, 2006.

26 G Maxwell Kelly. Coherence theorems for lax algebras and for distributive laws. In *Category seminar*, pages 281–375. Springer, 1974.

27 Wen Kokke and Ornela Dardha. Deadlock-free session types in linear Haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, pages 1–13, 2021.

28 Serge Lang. *Algebra*. Springer, New York, NY, 2002.

29 Gottfried Wilhelm Leibniz. Nova methodus pro maximis et minimis, itemque tangentibus, qua nec irrationals quantitates moratur. *Acta eruditorum*, 1684.

30 Sam Lindley and J Garrett Morris. A semantics for propositions as sessions. In *European Symposium on Programming Languages and Systems*, pages 560–584. Springer, 2015.

31 Sam Lindley and J Garrett Morris. Lightweight functional session types. *Behavioural Types: from Theory to Tools. River Publishers*, pages 265–286, 2017.

32 John Longley. When is a functional program not a functional program? In *ACM SIGPLAN Notices*, volume 34(9), pages 1–7. ACM, 1999.

33 John Longley. The sequentially realizable functionals. *Ann. Pure Appl. Log.*, 117(1-3):1–93, 2002. doi:10.1016/S0168-0072(01)00110-5.

34 Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5. Springer Science & Business Media, 2013.

35 Daniel Marshall and Dominic Orchard. Replicate, reuse, repeat: Capturing non-linear communication via session types and graded modal types. *Proceedings of PLACES 2022, Electronic Proceedings in Theoretical Computer Science*, 356:1–11, March 2022. doi:10.4204/eptcs.356.1.

36 Daniel Marshall, Michael Vollmer, and Dominic Orchard. Linearity and Uniqueness: An Entente Cordiale. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 346–375, Cham, 2022. Springer International Publishing.

37 Conor McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, pages 74–88, 2001.

38 Conor McBride. Clowns to the left of me, jokers to the right (pearl): Dissecting data structures. *SIGPLAN Not.*, 43(1):287–295, January 2008. doi:10.1145/1328897.1328474.

39 J. Garrett Morris. The best of both worlds: linear functional programming without compromise. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 448–461. ACM, 2016. doi:10.1145/2951913.2951925.

40 Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *International Workshop on Types for Proofs and Programs*, pages 252–267. Springer, 2004.

41 Dominic Orchard. Programming contextual computations. Technical report, University of Cambridge, Computer Laboratory, 2014.

42 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–30, 2019.

43 Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, Gothenburg, Sweden, September 1-3, 2014*, pages 123–135, 2014. doi:10.1145/2628136.2628160.

44 James Propp. Euler measure as generalized cardinality. *arXiv: Combinatorics*, 2002.

45 Ben Rudiak-Gould, Alan Mycroft, and Simon Peyton Jones. Haskell is not not ML. In *European Symposium on Programming*, pages 38–53. Springer, 2006.

46 Stephen H. Schanuel. Negative sets have Euler characteristic and dimension. In Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini, editors, *Category Theory*, pages 379–385, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

47 Rui Shi and Hongwei Xi. A linear type system for multicore programming in ATS. *Science of Computer Programming*, 78(8):1176–1192, 2013. doi:10.1016/j.scico.2012.09.005.

48 Kornel Szlachányi. Skew-monoidal categories and bialgebroids. *Advances in Mathematics*, 231(3-4):1694–1730, 2012.

49 Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 447–458, 2011. doi:10.1145/1926385.1926436.

50 Tarmo Uustalu, Niccolò Veltri, and Noam Zeilberger. The sequent calculus of skew monoidal categories. *Electronic Notes in Theoretical Computer Science*, 341:345–370, 2018.

51 Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, volume 3(4), page 5. Citeseer, 1990.

52 Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, 1992.

53 Philip Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 189–201, 2003.

54 Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.

55 David Walker. Substructural type systems. *Advanced Topics in Types and Programming Languages*, pages 3–44, 2005.

56 Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. Oxide: The essence of Rust, 2021. arXiv:1903.00982.

57 Nobuko Yoshida and Vasco T Vasconcelos. 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):73–93, 2007.

A Regular Linear Types

The type syntax for linear regular types (as discussed in Section 1) is as follows.

$$\tau ::= \tau \otimes \tau' \mid \tau \oplus \tau' \mid 1 \mid 0 \mid X \mid \mu X.\tau$$

where X ranges over recursion variables. We mostly focus on the non-recursive subset (just the first four constructs above), although recursive types make an appearance in Section 5 and we include them here for coherence with the usual description of regular types in the literature. Throughout we use τ and σ to range over types and also A, B, C, D.

The typing rules for linear regular types are as follows, which includes their standard term formers.

$$\frac{}{x : A \vdash x : A}\text{VAR} \quad \frac{\Gamma \vdash t : A \quad \Delta \vdash t' : B}{\Gamma, \Delta \vdash (t, t') : A \otimes B}\otimes\text{I} \quad \frac{\Gamma \vdash t : A \otimes B \quad \Delta, u : A, v : B \vdash t' : C}{\Gamma, \Delta \vdash \textbf{let } (u, v) = t \textbf{ in } t' : C}\otimes\text{E}$$

$$\frac{}{\vdash * : 1}1\text{I} \quad \frac{\Gamma \vdash t : 1 \quad \Delta \vdash t' : C}{\Gamma, \Delta \vdash \textbf{let } () = t \textbf{ in } t' : C}1\text{E} \quad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{inl}\ t : A \oplus B}\oplus\text{I}_L \quad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{inr}\ t : A \oplus B}\oplus\text{I}_R$$

$$\frac{\Gamma \vdash t : A \oplus B \quad \Delta, u : A \vdash t' : C \quad \Delta, v : B \vdash t'' : C}{\Gamma, \Delta \vdash \textbf{case } t \text{ of } \mathsf{inl}\ u \to t' \mid \mathsf{inr}\ v \to t'' : C}\oplus\text{E}$$

Note that in the above we do not include the linear function space $\tau \multimap \tau'$ since we considered just the syntax of regular types in Section 1, but linear functions are used throughout the paper. Their introduction and elimination rules are:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B} \quad \frac{\Gamma \vdash t : A \multimap B \qquad \Delta \vdash t' : A}{\Gamma, \Delta \vdash t\ t' : B}$$

As stated in Section 1, (linear) regular types behave *like* a commutative semiring, i.e., $\otimes$ and $\oplus$ are both commutative and associative with 1 and 0 as their corresponding units, with distributivity, but all up to isomorphism.

$$\begin{array}{ccc} A \otimes (B \otimes C) \cong (A \otimes B) \otimes C & A \oplus (B \oplus C) \cong (A \oplus B) \oplus C & A \otimes (B \oplus C) \cong (A \otimes B) \oplus (A \otimes C) \\ A \otimes B \cong B \otimes A & A \oplus B \cong B \oplus A & (B \oplus C) \otimes A \cong (B \otimes A) \oplus (C \otimes A) \\ 1 \otimes A \cong A & A \oplus 0 \cong A & A \otimes 0 \cong 0 \end{array}$$

All of the above isomorphisms are witnessed by pairs of mutually inverse functions.

Regular types also permit a notion of (positive) exponent, with τ^a defined inductively as:

$$\tau^0 = 1 \qquad \tau^{a+1} = \tau \otimes \tau^a$$

The usual positive exponent laws then hold up to isomorphism via associativity and commutativity (and removal of units in the case of the leftmost isomorphism), for all $a \geq 0, b \geq 0$:

$$\tau^1 \cong \tau \qquad \tau^a \otimes \tau^b \cong \tau^{a+b} \qquad (\tau^a)^b \cong \tau^{ab} \qquad (\sigma \otimes \tau)^a \cong \sigma^a \otimes \tau^a$$

A.1 Equations

This calculus has equations for (bi)functoriality of $\otimes$ and $\oplus$:

$$\begin{array}{rlrl} id \otimes id &= id & id \oplus id &= id \\ (f \otimes g) \circ (h \otimes k) &= (f \circ h) \otimes (g \circ k) & (f \oplus g) \circ (h \oplus k) &= (f \circ h) \oplus (g \circ k) \end{array}$$

The following equations are on the interaction of cotupling, injections and the $\oplus$ bifunctor, which are subset of those from Gibbons [18] that are derivable for the coproduct part of linear regular types:

$$\begin{array}{rlrlrl} [f, g] \circ \mathsf{inl} &= f & [h \circ \mathsf{inl}, h \circ \mathsf{inr}] &= h & [f, g] \circ (h \oplus k) &= [f \circ h, g \circ k] \\ [f, g] \circ \mathsf{inr} &= g & [\mathsf{inl}, \mathsf{inr}] &= id & h \circ [f, g] &= [h \circ f, h \circ g] \end{array}$$

The usual axioms for a (lax) monoidal functor hold for the (contravariant) inverse functor. These axioms are as follows:

$$\begin{array}{ll} \mathtt{mmult} \circ (\mathtt{munit} \otimes id) \circ \lambda_i = \lambda^{\ominus 1} & : A^{-1} \multimap (1 \otimes A)^{-1} \\ \mathtt{mmult} \circ (id \otimes \mathtt{munit}) \circ \rho_i = \rho^{\ominus 1} & : A^{-1} \multimap (A \otimes 1)^{-1} \\ \mathtt{mmult} \circ (\mathtt{mmult} \otimes id) \circ \alpha_i = \alpha^{\ominus 1} \circ \mathtt{mmult} \circ (id \otimes \mathtt{mmult}) & : A^{-1} \otimes (B^{-1} \otimes C^{-1}) \multimap ((A \otimes B) \otimes C)^{-1} \end{array}$$

where α is associativity and $\lambda : 1 \otimes A \multimap A$ and $\rho : A \otimes 1 \multimap A$ (and their inverses λ_i and ρ_i) witness the unit properties of $\otimes$.

B Involution is an Isomorphism

We show that for all τ, then τ^{-1} is a sequentially realizable involution up to isomorphism, i.e., $(\tau^{-1})^{-1} \cong \tau$, with $\tau \multimap ((\tau \multimap 1) \multimap 1)$ implemented as $\lambda x.\lambda f.f\,x$ (see Section 6) and the converse direction as follows, using the syntax of GV calculus (as formulated by [30]) rather than Granule as shown in Section 6, of type $((\tau \multimap 1) \multimap 1) \multimap \tau$:

$$(\lambda k.\mathbf{let}\ (x, c) = \mathsf{recv}(\mathsf{fork}(\lambda c.k(\lambda x.\mathsf{send}\ c\ x)))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x)$$

In order to prove that τ^{-1} is an involution up to isomorphism, we need to show that the functions $i : \tau \multimap ((\tau \multimap 1) \multimap 1$ and $j : ((\tau \multimap 1) \multimap 1) \multimap \tau$ are mutually inverse, or in other words that $j(i(t)) = t : \tau$ and $i(j(h)) = h : (\tau \multimap 1) \multimap 1$.

We leverage the $\beta\eta$-equality theory of GV based on its operational semantics given by [30], which is the same operational semantics for channels implemented in Granule [42].

We show both directions separately, as follows:

$$\begin{aligned}
j(i(t)) &= (\lambda k.\mathbf{let}\ (x, c) = \mathsf{recv}(\mathsf{fork}(\lambda c.k(\lambda x.\mathsf{send}\ c\ x)))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x)(\lambda f.f\ t)\\
&= (\mathbf{let}\ (x, c) = \mathsf{recv}(\mathsf{fork}(\lambda c.(\lambda f.f\ t)(\lambda x.\mathsf{send}\ c\ x)))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x)\\
&= (\mathbf{let}\ (x, c) = \mathsf{recv}(\mathsf{fork}(\lambda c.(\lambda x.\mathsf{send}\ c\ x)t))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x)\\
&= (\mathbf{let}\ (x, c) = \mathsf{recv}(\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x)
\end{aligned}$$

Checking the typing of the inner expression, we have:

$$\begin{aligned}
\lambda c.\mathsf{send}\ c\ t &: \mathsf{Chan}(!\tau.\mathsf{end}_!) \multimap \mathsf{Chan}(\mathsf{end}_!)\\
\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t) &: \mathsf{Chan}(?\tau.\mathsf{end}_?)\\
\mathsf{recv}(\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t)) &: \tau \otimes \mathsf{Chan}(\mathsf{end}_?)
\end{aligned}$$

So we have the binding $(x, c) : \tau \otimes \mathsf{Chan}(\mathsf{end}_?)$. Applying the global configuration semantics of GV [30, Figure 4], we then get the following:

$$\begin{aligned}
&\mathbf{let}\ (x, c) = \mathsf{recv}(\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x)\\
(\textsc{Lift+Fork}) \rightsquigarrow\ &(\nu \mathsf{c})(\mathbf{let}\ (x, c) = \mathsf{recv}\ \mathsf{c}\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x) \mid (\mathsf{send}\ \mathsf{c}\ t)\\
(\textsc{Lift+Send}) \rightsquigarrow\ &(\nu \mathsf{c})(\mathbf{let}\ (x, c) = (t, \mathsf{c})\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x) \mid \mathsf{c}\\
(\textsc{LiftV}) \rightsquigarrow\ &(\nu \mathsf{c})(\mathbf{let}\ () = \mathsf{wait}\ \mathsf{c}\ \mathbf{in}\ t) \mid \mathsf{c}\\
(\textsc{Lift+Wait}) \rightsquigarrow\ &\mathbf{let}\ () = ()\ \mathbf{in}\ t\\
(\textsc{LiftV}) \rightsquigarrow\ &t
\end{aligned}$$

Thus, $j(i(t)) = t : \tau$ as required.

In the opposite direction of the isomorphism we then have, $h : (\tau^{-1})^{-1}$ with

$$\begin{aligned}
i(j(h)) &= (\lambda x.\lambda f.f\ x)(\mathbf{let}\ (x, c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x)\\
&= (\lambda f.f\ (\mathbf{let}\ (x, c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x))\\
&= \mathbf{let}\ (x, c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ (\lambda f.f\ x)
\end{aligned}$$

Similarly to the first case, we then have the inner typing:

$$\begin{aligned}
\lambda x.\mathsf{send}\ c\ x &: \tau \multimap \mathsf{Chan}(\mathsf{end}_!)\\
h(\lambda x.\mathsf{send}\ c\ x) &: \mathsf{Chan}(\mathsf{end}_!)\\
\mathsf{fork}(h(\lambda x.\mathsf{send}\ c\ x)) &: \mathsf{Chan}(?\tau.\mathsf{end}_?)\\
\mathsf{recv}(\mathsf{fork}(h(\lambda x.\mathsf{send}\ c\ x))) &: \tau \otimes \mathsf{Chan}(\mathsf{end}_?)
\end{aligned}$$

So, again, $(x, c) : \tau \otimes \mathsf{Chan}(\mathsf{end}_?)$.

Applying the global configuration semantics of GV [30, Figure 4], we get:

$$\mathbf{let}\ (x, c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ (\lambda f.f\ x)$$

$$(\textsc{Lift+Fork}) \rightsquigarrow (\nu \mathsf{c})\,(\mathbf{let}\ (x, c) = \mathsf{recv}\ \mathsf{c}\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ (\lambda f.f\ x) \mid h\ (\lambda x.\mathsf{send}\ \mathsf{c}\ x))$$

Recall that $h : (\tau \multimap 1) \multimap 1$ therefore we know that h must necessarily use the input parameter, applying it to some $t : \tau$, therefore after some reduction in h $(\lambda x.\mathsf{send}\ \mathsf{c}\ x)$ we get some $\mathbf{let}\ c' = \mathsf{send}\ \mathsf{c}\ t\ \mathbf{in}\ h' : ()$ and some configuration C in the case that evaluating to this point had some other communication effects. Note that c' is not in the free variables of h' since the session typing tells us it is unused, i.e. $c' : \mathsf{Chan}(\mathsf{end}_!)$.

Then we get the continuing reduction sequence:

$$\begin{aligned}
(\textsc{LiftV}*) &\rightsquigarrow^* (\nu \mathsf{c})\,(\mathbf{let}\ (x, c) = \mathsf{recv}\ \mathsf{c}\ \mathbf{in}\ \mathbf{let}() = \mathsf{wait}\ c\,\mathbf{in}(\lambda f.f\ x) \mid \mathbf{let}\ c' = \mathsf{send}\ \mathsf{c}\ t\ \mathbf{in}\ h' \mid C)\\
(\textsc{Lift+Send}) &\rightsquigarrow (\nu \mathsf{c})\,(\mathbf{let}\ (x, c) = (t, \mathsf{c})\ \mathbf{in}\ \mathbf{let}() = \mathsf{wait}\ c\,\mathbf{in}(\lambda f.f\ x) \mid \mathbf{let}\ c' = \mathsf{c}\ \mathbf{in}\ h' \mid C)\\
(\textsc{LiftV}) &\rightsquigarrow (\nu \mathsf{c})\,(\mathbf{let}\ () = \mathsf{wait}\ \mathsf{c}\ \mathbf{in}\ (\lambda f.f\ t) \mid \mathbf{let}\ c' = \mathsf{c}\ \mathbf{in}\ h' \mid C)\\
(\textsc{Lift+Wait}) &\rightsquigarrow (\lambda f.f\ t) \mid h' \mid C
\end{aligned}$$

The result is a term that behaves like the original h; applying the term t from inside h to the continuation to f results in a configuration C and has some remaining reduction to do as h'. Thus $i(j(h)) = h : (\tau \multimap 1) \multimap 1$ as required.