Linearity, Uniqueness, Ownership: An Entente Cordiale

POPL 2022 *Student Research Competition* Extended Abstract

DANIEL MARSHALL, University of Kent, United Kingdom

1 PROBLEM AND MOTIVATION

Substructural type systems [15], which restrict the use of weakening and contraction rules from intuitionistic logic, are growing in popularity because they allow for a resourceful interpretation of data which can be used to rule out various software bugs. Indeed, substructurality is finally taking hold in modern programming; Haskell now has linear types as of GHC 9.0.1 [2] which are roughly based on Girard's linear logic [6] but integrated via graded function arrows, Clean has uniqueness types designed to ensure that values have at most a single reference to them [5, 11], and Rust offers an intricate ownership system for guaranteeing memory safety [9].

But despite this broad range of resourceful type systems, there is comparatively little understanding of their relative strengths and weaknesses. In particular, are linearity and uniqueness essentially the same thing, are they somehow 'dual' to one another, or is the truth somewhere in between? This work formalises the relationship between these two well-studied but rarely contrasted ideas, building on two distinct bodies of literature, and clearing up the confusion to answer these questions once and for all. We show that it is possible and advantageous to track the usage of resourceful data through linearity and also safely mutate values with unique references, by having both linear and unique types together in one unified system.

Furthermore, we study the guarantees provided by the resulting calculus and develop a practical implementation in the graded modal setting of the Granule language [10], adding a third kind of modality alongside coeffect and effect modalities. In future work, we will go on to discuss how just as Granule's coeffect system can allow for more fine-grained quantitative reasoning about resource usage, we can track references more precisely through a graded system based on fractional permissions [3], which links closely to the notion of ownership à la Rust.

2 BACKGROUND AND RELATED WORK

Reading the literature one might get the impression that linear types and uniqueness types are two names for the same concept, perhaps separated only by some minor implementational details. Indeed, the section on substructural type systems in *Advanced Topics in Types and Programming Languages* [15] describes uniqueness types as "*a variant of linear types*". But reading a different set of papers might give the contrasting impression that linearity and uniqueness are in some sense "dual" to one another, and so have considerably different behaviour for at least some applications. A recent publication on linear types for Haskell [2] describes the two concepts as being "*at their core, dual*", though this is later clarified to be only a "*weak duality*".

It is clear, at least, that both linear types and uniqueness types are *substructural* type systems, in that they both restrict the application of the structural rules (contraction and weakening, and in other settings possibly exchange) found in type systems that are the Curry-Howard counterparts to regular intuitionistic logic. This captures the well-known maxim that "*not all things in life are free*" [14]; many kinds of data behave *resourcefully*, and are subject to constraints on their usage. Sensitive data should not be infinitely duplicated and passed around freely, file handles should not

ACM student member number: 2563926. Entry for POPL'22 SRC, Graduate Category. Advised by Dominic Orchard.
Author's address: Daniel Marshall, University of Kent, Canterbury, Kent, United Kingdom, dm635@kent.ac.uk.

be arbitarily discarded without being properly closed, and array references should not be shared and allowed to be mutated while still in use, to name a few!

Linearity and uniqueness remain indistinguishable, however, until we introduce *unrestricted* values which do allow for contraction and weakening. Intuitively, if it is never possible to duplicate any value (so all values are linear), then it is also not possible for a value to have multiple references (so all values are unique) [12, 13]. Offering the capability to move between substructural and Cartesian (unrestricted) values is when differences between linear and unique types begin to arise [6, 7], as we shall now see when we describe how to represent both in a single type system.

3 APPROACH AND NOVELTY

The type system we will define, which we call the Linear-Cartesian-Unique calculus, builds on (intuitionistic multiplicative exponential) linear logic as its basis, with additional rules for a new uniqueness modality inspired by Harrington's uniqueness logic [7].

Values wrapped in the comonadic ! modality from linear logic can be extracted to retrieve a linear value, but once a value is restricted to behave linearly, we cannot remove this restriction. Thus, values with a *linear* type must be used exactly once, and cannot be duplicated or discarded. This makes them useful for data like *file handles*, since they cannot be copied and must be properly closed rather than simply thrown away.

In contrast, Harrington's uniqueness logic offers a monadic 'non-unique' modality $\circ$, and so our knowledge about a unique value is slightly different. We cannot say that a unique value will never be duplicated, since we can apply the 'return' of the monad to obtain a non-unique value which we may use freely. But unlike with a linear value we can guarantee that a unique value has never been duplicated previously, since it is not possible to escape the non-uniqueness monad. Hence uniqueness types are more useful for data such as *mutable arrays*, since we must guarantee our reference to the array is unique in order to safely mutate it in memory.

For our unified calculus, the crucial insight is that we can *treat non-linearity and non-uniqueness as the same state*, both represented by the ! modality (pronounced "bang"), since both cases allow for unrestricted behaviour with no guarantees about the past or restrictions on the future. Linear types are the basis of the calculus and behave according to the usual rules. We introduce a $*$ modality (pronounced "star", with syntax intended to be familiar to users of Clean) to represent unique values, with the important rules for this modality's behaviour given below.

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \&t : !A}\ \textsc{Borrow} \qquad \frac{\Gamma_1 \vdash t_1 : !A \quad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \text{copy } t_1 \text{ as } x \text{ in } t_2 : !B}\ \textsc{Copy} \qquad \frac{\varnothing \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{Nec}$$

The Borrow rule[1] maps a unique value to a non-linear one, allowing a guarantee of uniqueness to be forgotten just as we expect. The Copy rule says that a non-linear value can be copied to produce a unique one which can be used internally; the input is required to be non-linear so that we cannot circumvent a linearity restriction by copying a linear value, and the output is required to be non-unique so that we cannot leverage our temporary uniqueness to smuggle out a value which we can continue to use uniquely outside the context of the Copy. (These rules are accompanied by a necessitation rule, allowing values to be unique by default as long as they have no dependencies.)

Note that these two rules suggest a certain relationship between the two modalities in our calculus; if we ignore the presence of the $*$ modality, we see that ! acts much like a monad (similarly to $\circ$), with the Borrow rule having the shape of a monadic 'return' and the Copy rule acting as the

[1]The name Borrow is intended to evoke the concept of borrowing as in Rust, and in particular an *immutable borrow*, where a value can no longer be mutated while it is being shared [16]. Representing *mutable borrows* which only temporarily invalidate the original unique reference would require more fine-grained fractional tracking of how widely references have been shared, which is future work.

'bind'. Formally, $*$ is a functor over which ! becomes a *relative monad* [1]. Indeed, the following equalities hold on the uniqueness fragment of the calculus, which are exactly the required axioms:

$$\text{copy } t \text{ as } x \text{ in } \&x \equiv t \quad \text{copy } \&t \text{ as } x \text{ in } t' \equiv [t/x]t' \qquad \text{(unit laws)}$$

$$\text{copy } t_1 \text{ as } x \text{ in } (\text{copy } t_2 \text{ as } y \text{ in } t_3) \equiv \text{copy } (\text{copy } t_1 \text{ as } x \text{ in } t_2) \text{ as } y \text{ in } t_3 \quad (x \# t_3) \quad \text{(associativity)}$$

The implementation of uniqueness types in Granule follows much the same pattern as the rules from the calculus. Granule already possesses a *semiring graded necessity modality*, where for a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$, there is a family of types $\{\Box A_r\}_{r \in \mathcal{R}}$. We represent the ! from linear logic (and our extended calculus) via the pre-ordered semiring $\{0, 1, \omega\}$[2] with $!A = \Box A_\omega$. This semiring allows us to represent both linear and non-linear use via grades of 1 and ω.

Our implementation provides the uniqueness modality $*$ as in the calculus, which wraps a value that behaves substructurally as with linear values; the key difference is that we provide operations with syntactic sugar corresponding to the calculus, allowing ! to act monadically over unique values. A simple example of linear and unique types in action follows, to demonstrate the idea.

```
desire : !Cake → (Cake, Happy)
desire lots = let !cake = lots in (have cake, eat cake)

sip : *Coffee → (!Coffee, Awake)
sip fresh = let !coffee = &fresh in ([keep coffee], drink coffee)
```

Granule

In the first example, we are only able to have our cake and eat it too because the cake is *non-linear*; eliminating the ! modality gives us access to an unrestricted variable representing an unlimited amount of cake. The second example shows that borrowing (`&`) can convert a *unique* cup of coffee into an unrestricted one, so that it can be duplicated and used twice for the two separate functions. Note however that the uniqueness guarantee is lost in the process, so both of the output values are non-unique (though they may or may not be linear). We can continue to sip our unrestricted coffee as many times as we like, but we cannot try to pretend that it is still fresh after the first sip!

We also provide an interface for unique arrays of floating point numbers in Granule, offering primitives for creating, reading from, writing to, calculating the length of, and dereferencing arrays. Note that it is possible for our `writeFloatArray` primitive to update an array destructively in place since we have a guarantee that no other references exist to the array which has been passed in; this set of primitives allows us to evaluate the performance of our implementation, by measuring the performance gains from allowing for in-place updates.

4 RESULTS AND CONTRIBUTIONS

We formalise the relationship between linearity and uniqueness, and prove key properties for both resource conservation and uniqueness preservation, building a theoretical underpinning for a unified system of substructural types. This is the first step on the road towards properly understanding the relationships between more advanced substructural type systems, such as the fine-grained resource tracking of languages like Granule [10] and Idris [4] and the complex memory management provided by Rust [16]. Moreover, we have implemented this system in the graded modal setting of the Granule language, and have also carried out benchmarks to demonstrate the efficiency gains offered by making use of unique mutable arrays in Granule programs. It is apparent that incorporating uniqueness types into languages outside of Granule, such as via adding a new flavour of multiplicity to Haskell's linear type system [2], has the potential to offer similar benefits.

[2]It may not seem obvious that this modality does exactly represent the behaviour of !, and in fact capturing linear exponentials precisely does require some additional semiring structure which is present in Granule; this is discussed further in [8].

REFERENCES

[1] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2015. Monads Need Not be Endofunctors. *Logical Methods in Computer Science* 11, 1 (Mar 2015). https://doi.org/10.2168/lmcs-11(1:3)2015

[2] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158093

[3] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72.

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

[5] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. 2008. Uniqueness Typing Simplified. In *Implementation and Application of Functional Languages*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 201–218.

[6] Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50, 1 (1987), 1 – 101. https://doi.org/10.1016/0304-3975(87)90045-4

[7] Dana Harrington. 2006. Uniqueness Logic. *Theoretical Computer Science* 354, 1 (2006), 24–41.

[8] Jack Hughes, Daniel Marshall, James Wood, and Dominic Orchard. 2021. Linear Exponentials as Graded Modal Types. In *5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021)*. Rome (virtual), Italy. https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271465

[9] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

[10] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (July 2019), 30 pages. https://doi.org/10.1145/3341714

[11] Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. 1994. Guaranteeing Safe Destructive Updates Through a Type System with Uniqueness Information for Graphs. In *Graph Transformations in Computer Science*, Hans Jürgen Schneider and Hartmut Ehrig (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 358–379.

[12] Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*, Vol. 3. Citeseer, 5.

[13] Philip Wadler. 1991. Is There a Use for Linear Logic? *SIGPLAN Not.* 26, 9 (May 1991), 255–273. https://doi.org/10.1145/115866.115894

[14] Philip Wadler. 1993. A Taste of Linear Logic. In *Mathematical Foundations of Computer Science 1993*, Andrzej M. Borzyszkowski and Stefan Sokołowski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 185–210.

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

[16] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2021. Oxide: The Essence of Rust. https://arxiv.org/abs/1903.00982