DANIEL MARSHALL

University of Kent

A Hitchhiker's Guide to Linearity

...and uniqueness

...and ownership...

a Lambda Days talk

Some data is unrestricted.

but...

some data is **resourceful.**

~~infinitely copiable~~

~~arbitrarily discardable~~

~~universally mutable~~

Linear types are like **cakes**.

You can only eat them **once**.

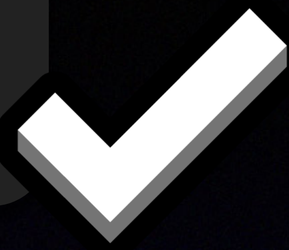
You **have** to eat them.

```
{-# LANGUAGE LinearTypes #-}

desire :: Cake ⊸ (Happy, Cake)
desire cake = (eat cake, have cake)
```

Linearity in Granule!


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

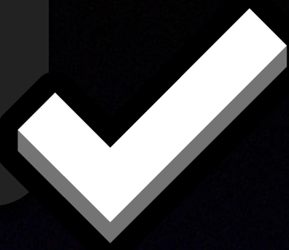


In practice (*file handles*)

```
linear : Char <IO>
linear = let
  h         ← openHandle ReadMode “towel.md”;
  (h’, c) ← readChar h
  in pure c
```

Linearity in Granule!


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



In practice (*file handles*)

```
linear : Char <IO>
linear = let
  h        ← openHandle ReadMode “towel.md”;
  (h’, c) ← readChar h;
  ()       ← closeHandle h’
  in pure c
```

Unique types are like **coffee.**

A fresh coffee has just been poured.

We can sip our coffee, **but...**

then it is **no longer** fresh!

```
share :: *Coffee → (Awake, *Coffee)
share coffee = (drink coffee, keep coffee)
```

"But what's the difference?"

Clean is a commercially developed, pure functional programming language. It uses *uniqueness types* (Barendsen and Smetsers, 1993), which are a variant of linear types, and strictness annotations (Nöcker and Smetsers, 1993) to help

Linear Haskell

Practical Linearity in a Higher-Order Polymorphic Language

JEAN-PHILIPPE BERNARDY, University of Gothenburg, Sweden
MATHIEU BOESPFLUG, Tweag I/O, France
RYAN R. NEWTON, Indiana University, USA
SIMON PEYTON JONES, Microsoft Research, UK
ARNAUD SPIWACK, Tweag I/O, France

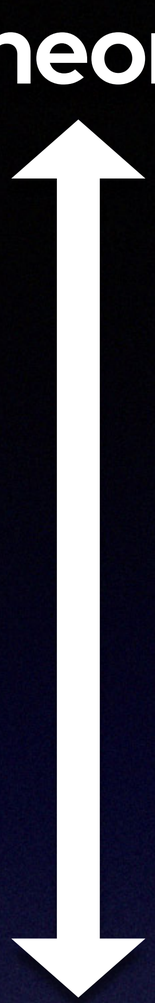
Linear type systems have a long and storied history, but not a clear p
guages such as OCaml or Haskell. In this paper, we study a linea
in mind: backwards-compatibility and code reuse acros
e benefits of linear types permeate conventio
to linear and non-linear counterparts, we inste
ceive inputs from linearly-bound values, but can
rate the efficacy of our linear type system — bo
lementation and how streamlined it makes it t
type system in GHC, the leading Haskell compil
s: mutable data with pure interfaces; and enforc

Linear types and uniqueness types are, at their core, dual: whereas a linear type is a contract that a function uses its argument exactly once even if the call's context can share a linear argument as many times as it pleases, a uniqueness type ensures that the argument of a function is not used anywhere else in the expression's context even if the callee can work with the argument as it pleases.

Unique types **guarantee** that a value has never been duplicated in the **past.**

Linear types **restrict** a value from ever being duplicated (or discarded) in the **future.**

A history of **linearity** and **uniqueness** (abridged)



Theory

Linear logic
Girard (1987)

Linear types can change the world!
Wadler (1990)

Linear Haskell: Practical linearity in a higher-order polymorphic language
Bernardy, Boespflug, Newton, Peyton Jones, Spiwack (2018)

Uniqueness logic
Harrington (2006)

Uniqueness typing simplified
de Vries, Plasmeijer, Abrahamson (2008)

Guaranteeing safe destructive updates through a type system with uniqueness information for graphs
Smetsers, Barendsen, van Eekelen, Plasmeijer (1994)

Practice

Uniqueness logic

Structural rules are restricted, as in linear logic...

...and the ◦ modality allows for discarding and duplication, as with ! from linear logic.

Unique **Cartesian** **Linear**

"How can we use both?"

Unique *a

↓ sharing

Cartesian !a

↓ dereliction

Linear a

Unique values under an **additional** * modality.

Cartesian values under a **comonadic** ! modality.

Linear values as the base.


```
share : ∀ {a : Type}
      . *a → !a
```

The **!** modality acts as a **relative monad** over ***.**

(Unit laws and associativity hold!)

```
clone : ∀ {a b : Type}
      . !a → (*a → !b) → !b
```

```
return :: Monad m ⇒ a → m a
```

```
bind :: Monad m ⇒ m a → (a → m b) → m b
```

"And how does this work in practice?"

Download and play!

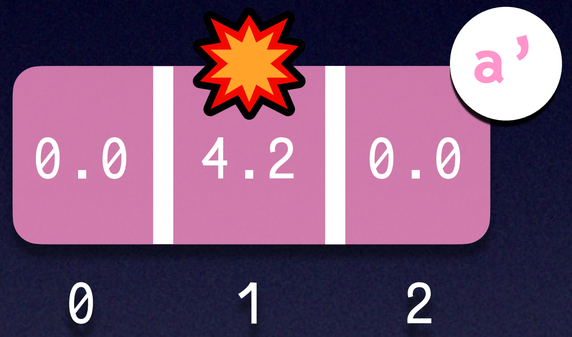
- Already has a **linear** base.
- Already represents **!** as a **coeffect** modality (dual to **effects**).
- Represent ***** as a **third** flavour of modality (called a **guarantee**).

https://granule-project.github.io

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

Uniqueness in Granule (*mutable arrays*)

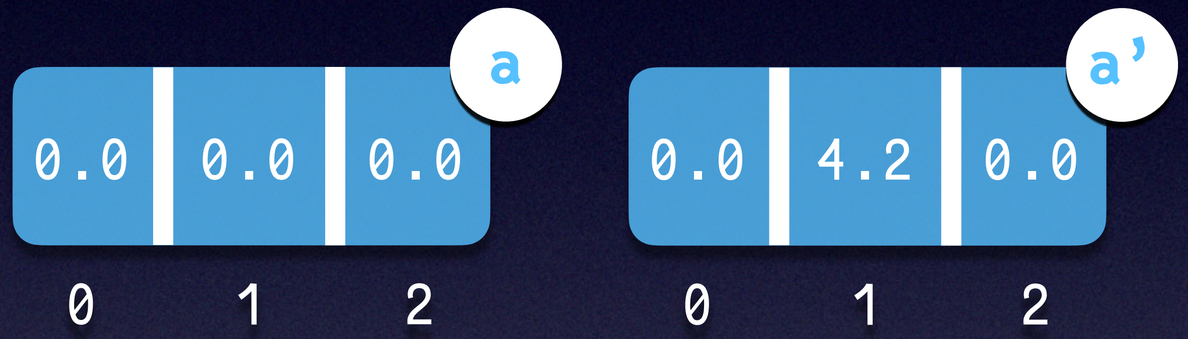

```
unique : (Float, *FloatArray)
unique = let
  a     = newFloatArray 3;
  a'    = writeFloatArray a 1 4.2
  in readFloatArray a' 1
```



mutated
in place!

Uniqueness in Granule (*immutable arrays*)


```
unique : (Float, FloatArray)
unique = let
  a    = newFloatArrayI 3;
  a'   = writeFloatArrayI a 1 4.2
  in readFloatArrayI a' 1
```



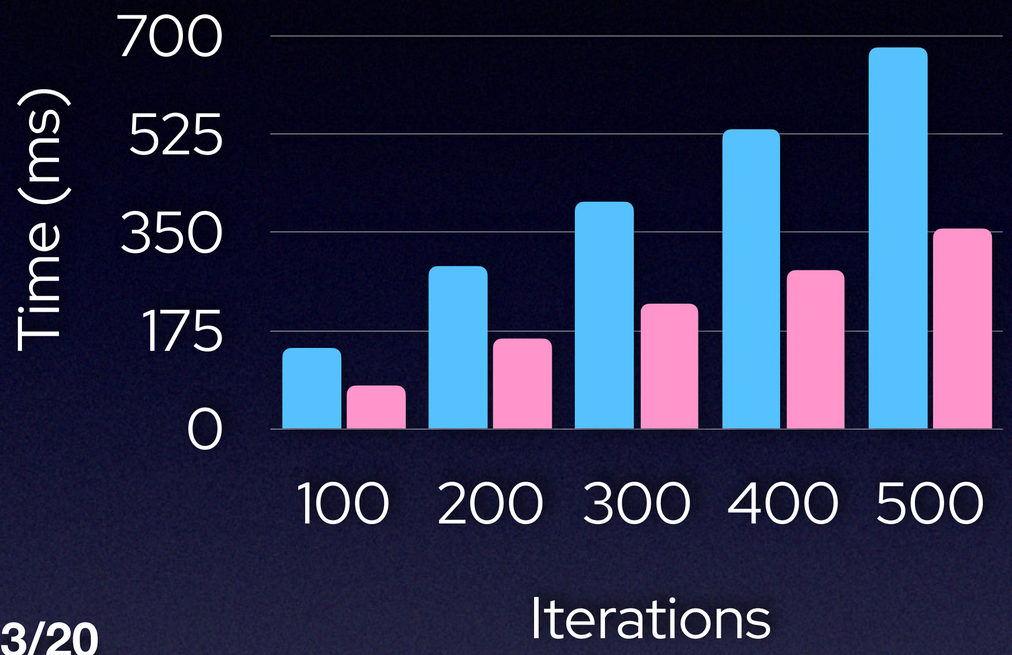
Uniqueness in Granule (*mix and match*)


```
unique : (Float, FloatArray)
unique = let
  a    = newFloatArray 3;
  a'   = writeFloatArray a 1 4.2;
  !a'' = share a'
  in readFloatArrayI a'' 1
```

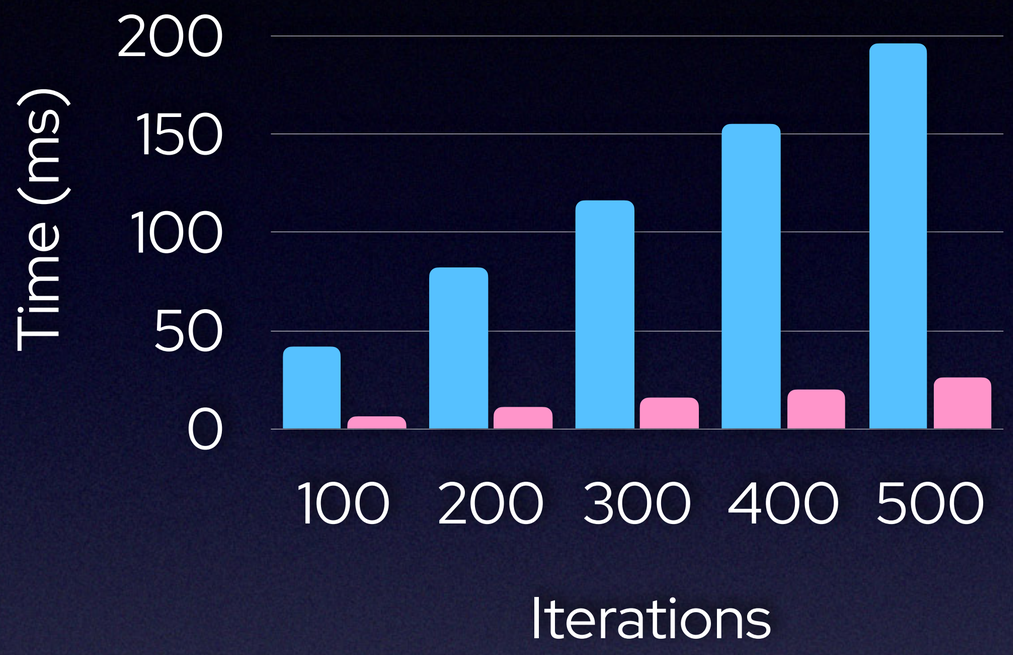
What advantages do **unique arrays** offer?

It's safe to mutate them in place, so uniqueness gives us some **performance benefits**!

Overall runtime



Garbage collection overhead



Linear Logic

$!A$ modality represents
non-linear usage of A

generalises to...

Bounded Linear Logic

$!_r A$ family of modalities where
r gives an **upper bound** on usage

generalises to...

Graded Modal Types

$\Box_r A$ family of modalities where r is drawn from a **pre-ordered semiring**

gr

$\Box_r A$ = A [r]

Graded modal types in action!


“none, one, tons”

```
desire : Cake [Many] → (Happy, Cake [Many])
```

Exact usage

```
desire : Cake [2] → (Happy, Cake [1])
```

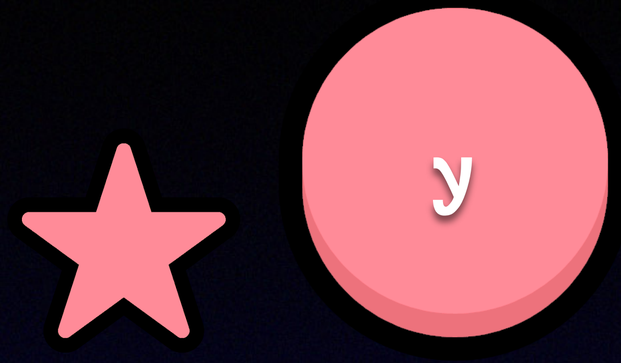
```
desire : ∀ {n : Nat} . Cake [n+1] → (Happy, Cake [n])
```

Intervals

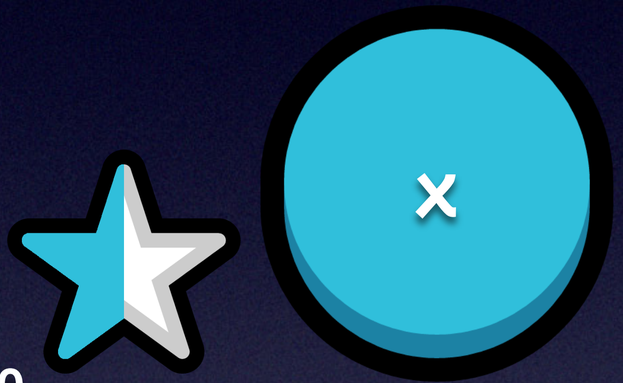
```
desire : Maybe Cake → Cake [0..1] → Happy
desire Nothing      [default] = eat default;
desire (Just cake) [default] = eat cake
```

Then what about **ownership**?

Ownership can be modelled by **capabilities.**



Borrowing splits capabilities into **fractions.**



Ownership and borrowing in Rust...

```
struct Colour(u32, u32, u32);
let granule = Colour(74, 109, 218);
```

```
let x = &granule;
let y = &granule; // ok!
```

```
let x = &granule;
let y = &mut granule; // error :(
```

```
let x = &mut granule;
let y = &mut granule; // error :(
```

```
let x = &mut granule;
let y = &mut *x; // ok!
```

Ownership in **Granule**!

Linear types can be generalised to allow for **quantitative** restrictions.

$!_3\ a \to !_2\ a \times a$

*(similar to **bounded linear logic**)*

Unique types can be generalised to allow for **fractional** guarantees.

$*a \longleftrightarrow \&_1\ a \longleftrightarrow \&_{1/2}\ a \times \&_{1/2}\ a \longleftrightarrow \&_{1/4}\ a \times \&_{1/4}\ a \times \&_{1/2}\ a$

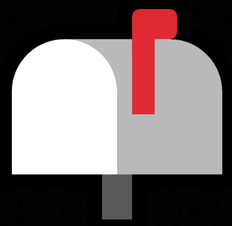
mutable borrow!

immutable borrows!

reborrowing!

*(similar to **fractional permissions**)*

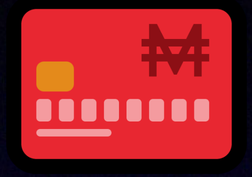
and another thing...



Communication

Session types for linear communication

Graded session types for non-linear communication



Security

Confidential data will never be leaked in the future

```
leak : Recipe [Private]
    → Recipe [Public]
```

Data with integrity has never been leaked in the past

```
forge : Recipe [Public]
     → Recipe *{Trusted}
```

Thank you for listening!

In summary:

Linearity restricts the future, uniqueness guarantees the past

Quantitative types generalise linear types, offering more precision

Ownership generalises uniqueness, offering more flexibility

Graded types generalise all of the above!

@starsandspirals

@daniel@types.pl

so long... and thanks for all the fish!

https://starsandspira.ls