Graded Types - Part 3

Communication, uniqueness, and mutability

Dominic Orchard, 24-28th July 2023, SPLV23

UNIVERSITY OF CAMBRIDGE
University of Kent

A few things re pattern matching and substructurality…

Generic deriving useful operations for graded + linear types

TLLA 2021

Deriving Distributive Laws for Graded Linear Types

Jack Hughes
School of Computing, University of Kent

Michael Vollmer
School of Computing, University of Kent

Dominic Orchard
School of Computing, University of Kent

The recent notion of graded modal types provides a framework for extending type theories with fine-grained data-flow reasoning. The Granule language explores this idea in the context of linear types. In this practical setting, we observe that the presence of graded modal types can introduce an additional impediment when programming: when composing programs, it is often necessary to 'distribute' data types over graded modalities, and vice versa. In this paper, we show how to automatically derive these distributive laws as combinators for programming. We discuss the implementation and use of this automated deriving procedure in Granule, providing easy access to these distributive combinators. This work is also applicable to Linear Haskell (which retrofits Haskell with linear types via grading) and we apply our technique there to provide the same automatically derived combinators. Along the way, we discuss interesting considerations for pattern matching analysis via graded linear types.

So far….

Data
as a resource

e.g.,

- Specifying programs based on (re)use
- Security / confidentiality information
- File handles with protocols of interaction

Session types in Granule

```
send        :  LChan (Send a p) -> a -> LChan p
recv        :  LChan (Recv a p) -> (a, LChan p)
forkLinear  :  (LChan p -> ()) -> LChan (Dual p)
close       :  LChan End -> ()

selectLeft  :  LChan (Select p1 p2) -> LChan p1
selectRight :  LChan (Select p1 p2) -> LChan p2
offer       :  (LChan p1 -> a) -> (LChan p2 -> a)
                 -> LChan (Offer p1 p2) -> a
```

From the exercises sheet…

3. Using the `Cake` module, define a function `mange` that takes $n + m$ cakes, consumes n of them, leaving m left and n amounts of happiness.

What does "n + m cakes" mean?

It depends on our reduction semantics

Graded modalities and reduction semantics

$$\frac{t \rightsquigarrow t'}{[t] \rightsquigarrow [t']}$$

$$[t] : \Box_2 A$$

"*two uses of an* A"

$\text{let } [x] = [v] \text{ in } t' \rightsquigarrow [v/x]t$

$\text{let } [x] = [t] \text{ in } t' \rightsquigarrow [t/x]t$

CBV

evaluate t once

$t \rightsquigarrow^* v$

use value v twice

CBN

evaluate t twice

$t \rightsquigarrow^* v$ $\quad$ $t \rightsquigarrow^* v$

Graded modalities and reduction semantics

$$[\mathsf{newArray}\ldots] : \Box_{2}\, A$$

"two uses of an A*"*

CBV

CBN

evaluate t once

$t \rightsquigarrow^{*} \mathsf{arrayRef}(i)$

evaluate t twice

$t \rightsquigarrow^{*} \mathsf{arrayRef}(i)$ $\qquad t \rightsquigarrow^{*} \mathsf{arrayRef}(i')$

arrayRef(i) could be used in two threads, creating a race on writes

Unsound and unsafe

CBV, grading, and resources

we wanted to
a real language,
with grades for
analysis

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

resourceAllocator(e)

A term e whose reduction to a normal form creates a heap-allocated resource, e.g., a channel, array, handle

$$\frac{}{\text{resourceAllocator}(\text{forkLinear } v)}$$

and other related resource allocators

+induct over syntax

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

$$\frac{\text{resourceAllocator}(t_1)}{\text{resourceAllocator}(t_1\ t_2)} \quad \frac{\text{resourceAllocator}(t_2)}{\text{resourceAllocator}(t_1\ t_2)} \quad \frac{\text{resourceAllocator}(t_1)}{\text{resourceAllocator}((\lambda x.t_1)\ t_2)}$$

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

$$\frac{\text{resourceAllocator}(t_1)}{\text{resourceAllocator}(\mathbf{let\ unit} = t_1\ \mathbf{in}\ t_2)} \quad \frac{\text{resourceAllocator}(t_2)}{\text{resourceAllocator}(\mathbf{let\ unit} = t_1\ \mathbf{in}\ t_2)}$$

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

$$\frac{\text{resourceAllocator}(t_1)}{\text{resourceAllocator}((t_1, t_2))} \quad \frac{\text{resourceAllocator}(t_2)}{\text{resourceAllocator}((t_1, t_2))}$$

CBV, grading, and resources

we wanted to a real language, with grades for analysis

$$\frac{\neg\text{resourceAllocator}(e) \qquad [\Gamma] \vdash t : A}{r * \Gamma \vdash [t] : \Box_r\, A}$$

But, you can return to the standard CBN theory:

```
language CBN
```

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

Recall: Types for the "four Rs" of PL design

Reading

- Documentation

'Riting

- Specification (intention)
- Synthesis

Reasoning

- Guarantee absence of some bugs
- Program properties (see 'Free Theorems')

Running

- Optimisations

In-place update (Mutation)

Dominic A. Orchard:
The four Rs of programming language design. Onward! 2011: 157-162

ESOP 2022

Linearity and Uniqueness: An Entente Cordiale

Daniel Marshall[1] (✉), Michael Vollmer[1], and Dominic Orchard[1,2]

[1] University of Kent, Canterbury, UK
{dm635,m.vollmer,d.a.orchard}@kent.ac.uk
[2] University of Cambridge, UK

Abstract. Substructural type systems 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 roughly based on Girard's linear logic but integrated via graded function arrows, Clean has uniqueness types designed to ensure that values have at most a single reference to them, and Rust has an intricate ownership system for guaranteeing memory safety. But despite this broad range of resourceful type systems, there is comparatively little understanding of their relative strengths and weaknesses or whether their underlying frameworks can be unified. There is often confusion about whether linearity and uniqueness are essentially the same, or are instead 'dual' to one another, or somewhere in between. This paper formalises the relationship between these two well-studied but rarely contrasted ideas,

Linear types are like cake

You can only eat them once.

You have to eat them

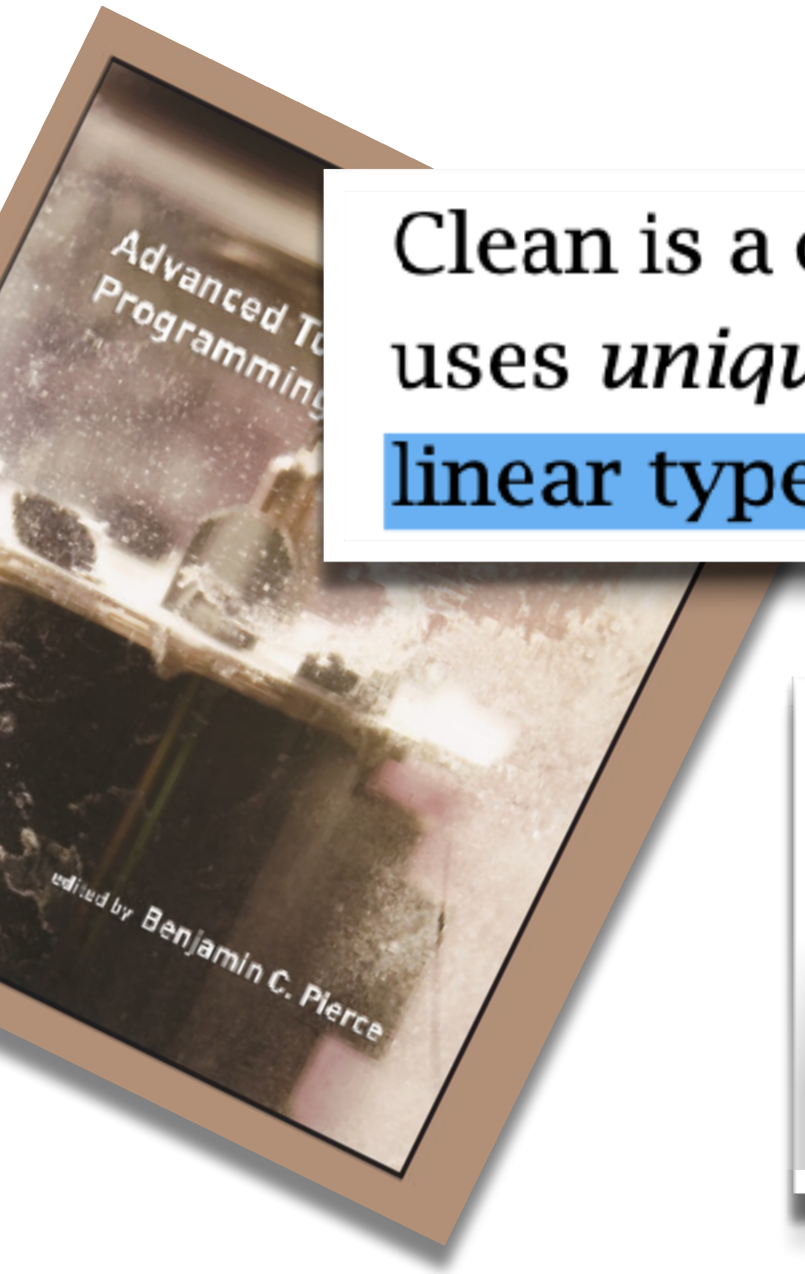
```
desire : Cake -> (Happy, Cake)
desire cake = (eat cake, have cake)
```

Unique types are like coffee

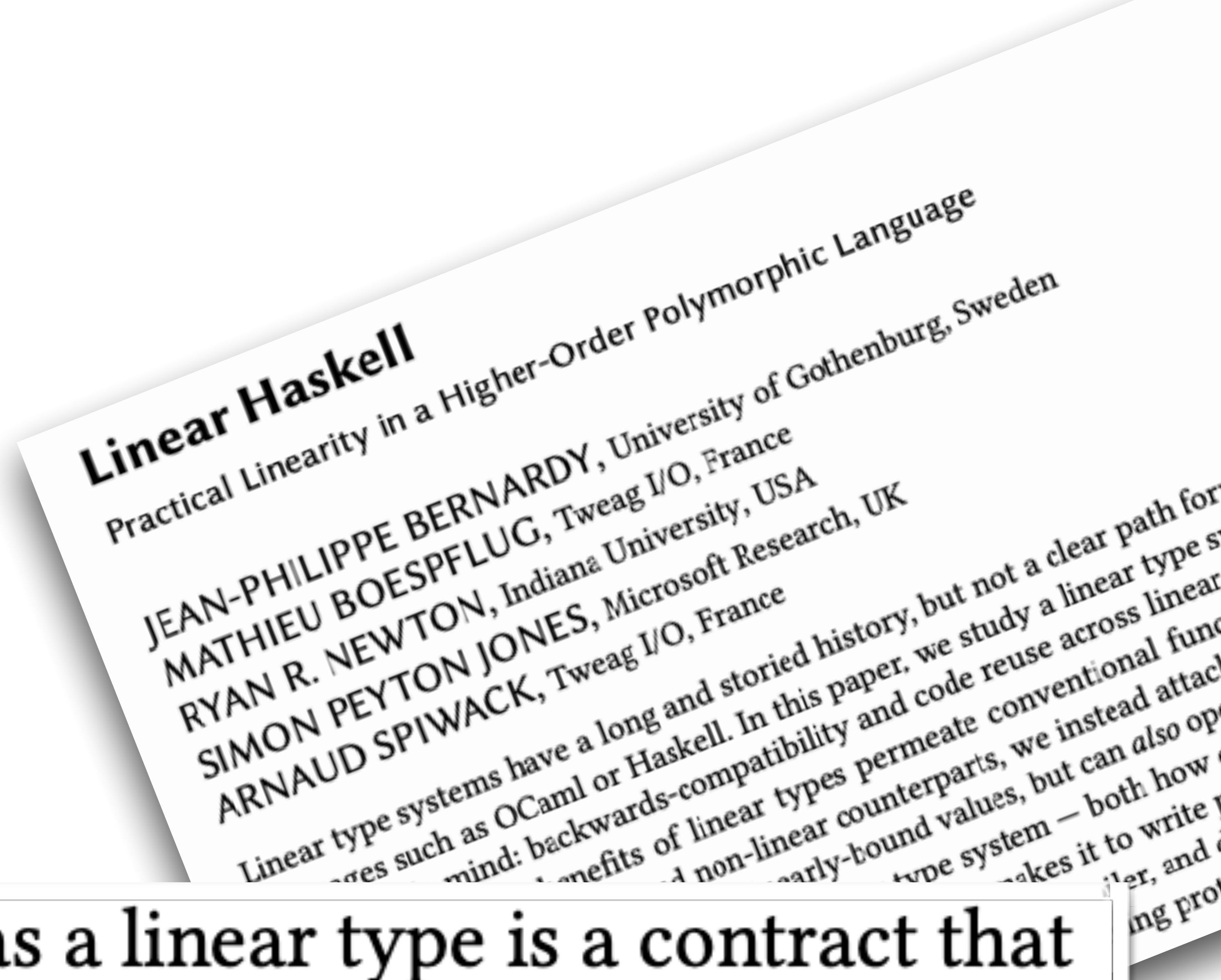
A fresh coffee has just been poured.
We can sip our coffee, but…
then it is no longer fresh!

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

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 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.**

“How can we use both?”

Unique — *a — **Unique** values under an **additional** modality *

↓ **sharing**

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

↓ **dereliction**

Linear — a — **Linear** values as the base

Mutable array interface (*on unique arrays*)

```
newFloatArray : Int -> *FloatArray

readFloatArray : *FloatArray -> Int -> (Float, *FloatArray)

writeFloatArray : *FloatArray -> Int -> Float -> *FloatArray

lengthFloatArray : *FloatArray -> (Int, *FloatArray)

deleteFloatArray : *FloatArray -> ()
```

and with…

$$\frac{}{\text{resourceAllocator}(\text{newFloatArray } v)}$$

Sharing ("*borrow*") and cloning

$$\frac{\Gamma \vdash t : {*}A}{\Gamma \vdash \mathsf{share}\ t : {!A}} \qquad \frac{\Gamma_1 \vdash t_1 : {!A} \quad \Gamma_2, x : {*}A \vdash t_2 : {!B}}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : {!B}}$$

Therefore $!A$ is a *relative monad* (relative to $*$)

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{return}\ t : MA} \qquad \frac{\Gamma_1 \vdash t_1 : {!A} \quad \Gamma_2, x : A \vdash t_2 : {!B}}{\Gamma_1 + \Gamma_2 \vdash (\mathsf{do}\ x \leftarrow t_1; t_2) : {!B}}$$

Relative monad axioms

(right-unit) $\mathsf{clone}\ t\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{share}\ x) \equiv t$

(left-unit) $\mathsf{clone}\ (\mathsf{share}\ v)\ \mathsf{as}\ x\ \mathsf{in}\ t' \equiv [v/x]t'$

(assoc) $$\mathsf{clone}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{clone}\ t_2\ \mathsf{as}\ y\ \mathsf{in}\ t_3) \equiv \mathsf{clone}\ (\mathsf{clone}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2)\ \mathsf{as}\ y\ \mathsf{in}\ t_3$$

(where $x \notin \mathsf{fv}(t_3)$)

Immutable array interface (*on linear arrays*)

```
newFloatArrayI : Int -> FloatArray

readFloatArrayI : FloatArray -> Int -> (Float, FloatArray)

writeFloatArrayI : FloatArray -> Int -> Float -> FloatArray

lengthFloatArrayI : FloatArray -> (Int, FloatArray)

deleteFloatArray : FloatArray -> ()
```

No delete as multiple references may be held in CBV

Graded uniqueness (*the third flavour…*)

Work under review 🕶

$$*A$$

$$\&_* A \qquad \&_p A \qquad \&_1 A$$

Uniquely owned A

$p \in (0,1) \subset \mathbb{Q}$

Immutable borrow

Mutable borrow

+ primitives for borrowing, mutable borrowing by splitting/joining lifetimes

e.g. split : $\&_p A \multimap \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A$

(follow Daniel Marshall's work -> https://starsandspira.ls/)

Here is what I consider one of the biggest mistakes of all in modal logic: concentration on a system with just one modal operator

Dana Scott (1968)