



Granule: A general purpose-
language for fine-grained reasoning

past, present, and a possible future

$$(\forall m. \square_{-m} A) \wedge A \wedge \diamond(\exists n. \square_n A)$$





research

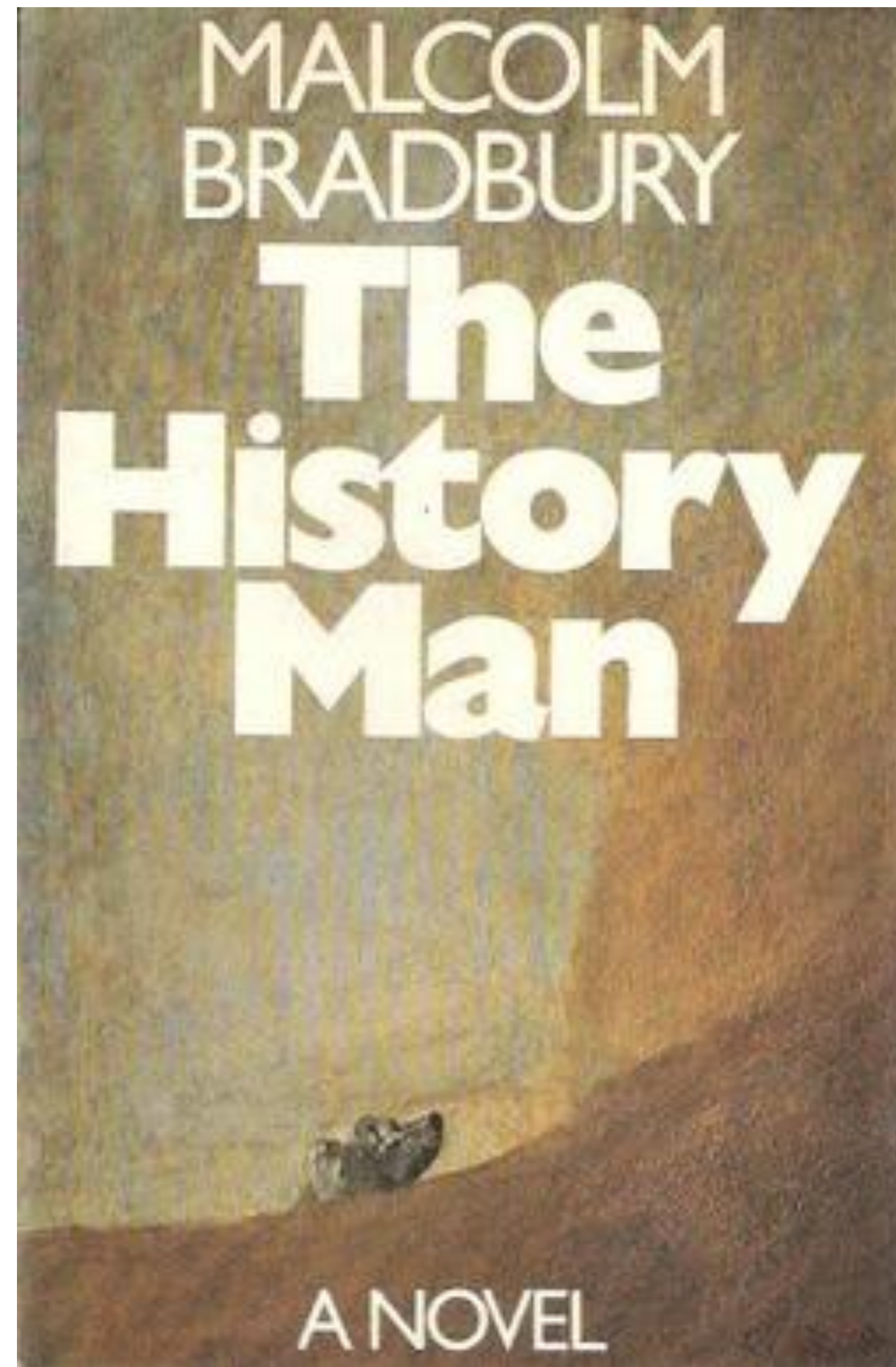
Granule: A ~~general purpose-~~
language for fine-grained reasoning

past, present, and a possible future

$$(\forall m. \square_{-m} A) \wedge A \wedge \diamond(\exists n. \square_n A)$$



The past



Coeffects: Unified static analysis of context-dependence *

Tomas Petricek, Dominic Orchard and Alan Mycroft

University of Cambridge, UK
 {tp322, dao29, am}@cl.cam.ac.uk

Abstract. Monadic effect systems provide a unified way of tracking effects of computations, but there is no unified mechanism for tracking how computations rely on the environment in which they are executed. This is becoming an important problem for modern software – we need to track where distributed computations run, which resources a program uses and how they use other capabilities of the environment.

We consider three examples of context-dependence analysis: *liveness* analysis, tracking the use of *implicit parameters* (similar to tracking of *resource usage* in distributed computation), and calculating caching requirements for *dataflow* programs. Informed by these cases, we present a unified calculus for tracking context dependence in functional languages together with a categorical semantics based on *indexed comonads*. We believe that indexed comonads are the right foundation for constructing context-aware languages and type systems and that following an approach akin to monads can lead to a widespread use of the concept.

Modern applications run in diverse environments – such as mobile phones or the cloud – that provide additional resources and meta-data about provenance and security. For correct execution of such programs, it is often more important to understand how they *depend* on the environment than how they *affect* it.

Understanding how programs affect their environment is a well studied area: *effect systems* [13] provide a static analysis of effects and *monads* [8] provide a unified semantics to different notions of effect. Wadler and Thiemann unify the two approaches [17], *indexing* a monad with effect information, and showing that the propagation of effects in an effect system matches the semantic propagation of effects in the monadic approach.

No such unified mechanism exists for tracking the context requirements. We use the term *coeffect* for such contextual program properties. Notions of context have been previously captured using *comonads* [14] (the dual of monads) and by languages derived from *modal logic* [12,9], but these approaches do not capture

Coeffects: A calculus of context-dependent computation

Tomas Petricek Dominic Orchard Alan Mycroft

University of Cambridge
 {firstname.lastname}@cl.cam.ac.uk

Abstract

The notion of *context* in functional languages no longer refers just to variables in scope. Context can capture additional properties of variables (usage patterns in linear logics; caching requirements in dataflow languages) as well as additional resources or properties of the execution environment (rebindable resources; platform version in a cross-platform application). The recently introduced notion of *coeffects* captures the latter, whole-context properties, but it failed to capture fine-grained per-variable properties.

We remedy this by developing a generalized coeffect system with annotations indexed by a coeffect *shape*. By instantiating a concrete shape, our system captures previously studied *flat* (whole-context) coeffects, but also *structural* (per-variable) coeffects, making coeffect analyses more useful. We show that the structural system enjoys desirable syntactic properties and we give a categorical semantics using extended notions of *indexed comonad*.

The examples presented in this paper are based on analysis of established language features (liveness, linear logics, dataflow, dynamic scoping) and we argue that such context-aware properties will also be useful for future development of languages for increasingly heterogeneous and distributed platforms.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms Languages, Theory

Keywords Context; Types; Coeffects; Indexed comonads

1. Introduction

Context is important for defining meaning – not just in natural languages, but also in logics and programming languages. The standard notion of context in programming is an environment providing values for free variables. An open term with free variables is context dependent – its meaning depends on the free-variable context. The simply-typed λ -calculus famously analyses such context usage. Other systems go further. For example, bounded linear logic tracks the number of times a variable is used [7].

In this paper, we develop a calculus for capturing various notions of context in programming. A key feature and contribution of the calculus is its *coeffect system* which provides a static analysis for contextual properties (coeffects). The system follows the style of type and effect systems, but captures a different class of properties. Another key contribution of the calculus is its semantics which can be smoothly instantiated for specific notions of context.

Coeffect systems were previously introduced as a generic analysis of context dependence which can be instantiated for various notions of context [15]. The formalization was restricted to tracking a class of *whole-context* properties where terms have just one coeffect. This limited the applications and precision of any analysis. For example, a whole-context liveness analysis marks the entire free-variable context as live (some variable may be used) or dead (no variable is used), but it cannot record liveness *per variable*.

We develop a more general system which captures both per-variable coeffects, which we call *structural*, and whole-context coeffects, which we call *flat*, and more. Our key contributions are:

- We present the *coeffect calculus* which augments the simply-typed λ -calculus with a general coeffect type system (Section 3). We demonstrate the two classes of flat (whole context) and structural (fine-grained, per variable) systems.
- We show practical examples, instantiating the calculus for structural systems capturing variable usage based on bounded linear logic, dataflow caching, and precise liveness analysis. We also instantiate the calculus to flat systems, building on and extending previous examples from [15].
- We discuss the syntactic properties of flat and structural variants of the coeffect calculus (Section 4). Notably, structural systems satisfy type preservation under both β -reduction and η -expansion, allowing their use with both call-by-name and call-by-value languages. This important property distinguishes structural coeffects from both effect systems and flat coeffects.
- We provide a denotational semantics, revisiting and extending the notion of *indexed comonads* to the structural setting (Section 5). We prove soundness by showing the correspondence

Type-and-coeffect systems

$$C @ \Gamma \vdash e : \tau$$

flat
ambient
“whole context”

structural
“per-variable”

$$\{\text{console}\} @ \Gamma \vdash \text{print "wat"} : \text{unit}$$

$$\langle 1, 0 \rangle @ x : A, y : B \vdash x : A$$

Modelled by (exponential) graded comonads

$$D : (M, \bullet, 1) \rightarrow [\mathcal{C}, \mathcal{C}]$$

$$\varepsilon_A : A \rightarrow D1A$$

$$\delta_{r,s,A} : D(r \bullet s)A \rightarrow DrDsA$$

Combining Effects and Coeffects via Grading

Marco Gaboardi
SUNY Buffalo, USA
gaboardi@buffalo.edu

Shin-ya Katsumata
Kyoto University, Japan
sinya@kurims.kyoto-u.ac.jp

Dominic Orchard
University of Cambridge, UK
University of Kent, UK
dominic.orchard@cl.cam.ac.uk

Flavien Breuvert
Inria Sophia Antipolis, France
flavien.breuvert@inria.fr

Tarmo Uustalu
Institute of Cybernetics at TUT, Estonia
tarmo@cs.ioc.ee

Abstract

Effects and *coeffects* are two general, complementary aspects of program behaviour. They roughly correspond to computations which change the execution context (effects) versus computations which make demands on the context (coeffects). Effectful features include partiality, non-determinism, input-output, state, and exceptions. Coeffectful features include resource demands, variable access, notions of linearity, and data input requirements.

The effectful or coeffectful behaviour of a program can be captured and described via type-based analyses, with fine grained information provided by monoidal effect annotations and semiring coeffects. Various recent work has proposed models for such typed calculi in terms of *graded (strong) monads* for effects and *graded (monoidal) comonads* for coeffects.

Effects and coeffects have been studied separately so far, but in practice many computations are both effectful and coeffectful, *e.g.*, possibly throwing exceptions but with resource requirements. To remedy this, we introduce a new general calculus with a combined *effect-coeffect system*. This can describe both the *changes* and *requirements* that a program has on its context, as well as interactions between these effectful and coeffectful features of computation. The effect-coeffect system has a denotational model in terms of effect-graded monads and coeffect-graded comonads where interaction is expressed via the novel concept of *graded distributive laws*. This graded semantics unifies the syntactic type theory with the denotational model. We show that our calculus can be instantiated to describe in a natural way various different kinds of interaction between a program and its evaluation context.

1. Introduction

Pure, total functional programming languages are highly amenable to clear and concise semantic descriptions. This semantics aids both correct-by-construction programming and tools for reasoning about program properties. A pure program can be described as a mathematical object that is isolated from the real world. However, even in the most abstract setting, a program is hardly isolated. Instead it interacts with its evaluation context; *paradise is lost*.

The interaction of a program with its context can be described in several ways. For instance it can be described by recording the *changes* that a program performs on its context, *e.g.* the program can write to a memory cell or it can print a character on an output display. At the same time the interaction can also be expressed by recording the *requirements* that a program has with respect to its context, *e.g.* the program can require a given amount of memory or to read the input from some channel. These two aspects correspond to the view of a program as a *producer* and as a *consumer*.

Computational effects and monads The need to describe the interaction of a program with its context emerged early in pure functional programming. Indeed, basic operations like input-output are inconceivable in a program that runs in isolation. Most of the original efforts to understand the interaction of a program with its context focussed on input-output, stateful computations, non-determinism, and probabilistic behaviours. This leads to the distinction between pure and *effectful* computation. The diverse collection of interactions described above are often referred to as *computational effects*. For our presentation, we identify these with behaviours that change the execution context, or as *producer effects*.

Linear-lambda calculus as basis

$$A \multimap B$$

Coeffects via graded comonads

$$\square_r A$$

Effects via graded monads

$$\diamond_f A$$

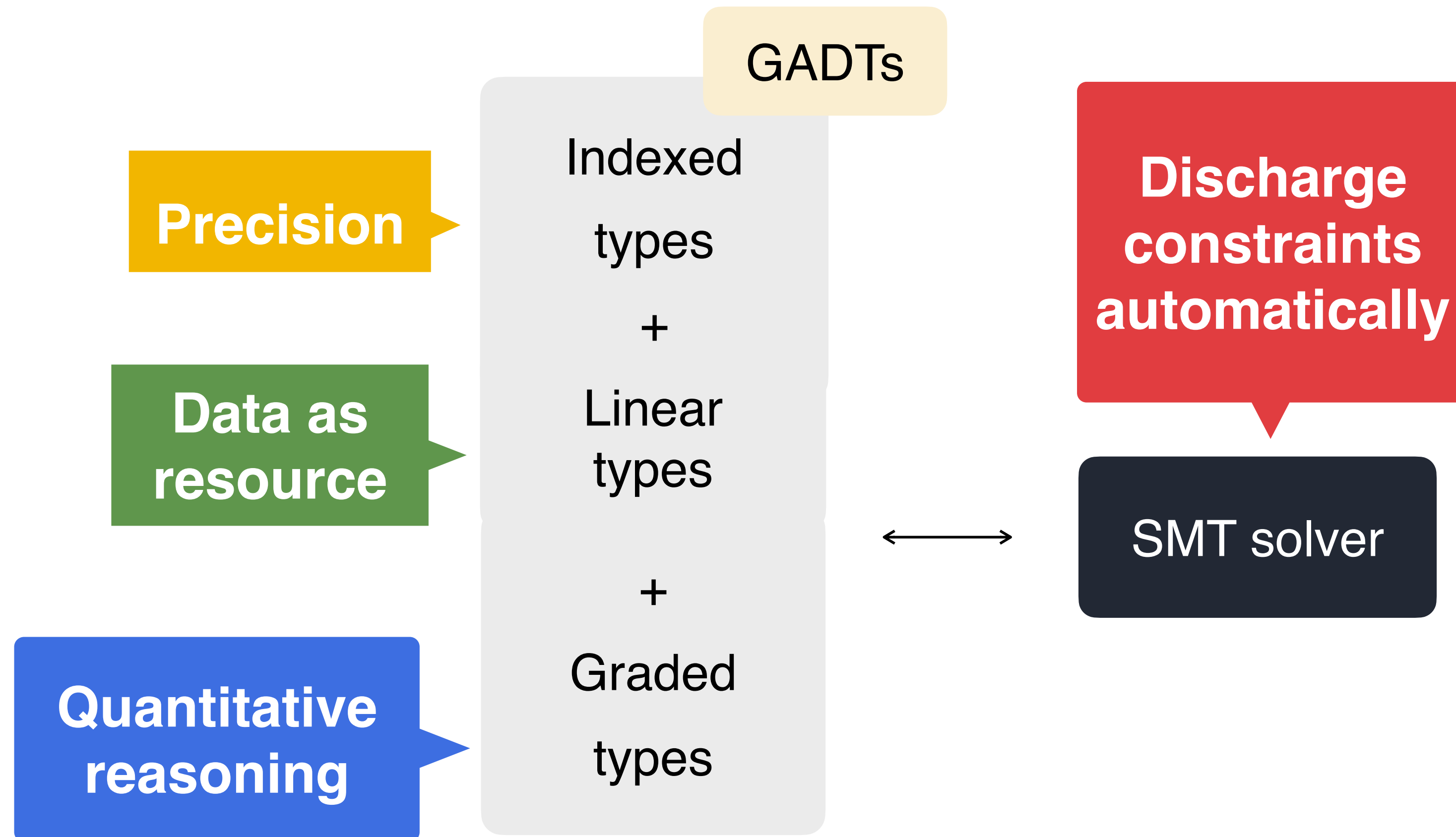
2017

Graded Types
Theory

Practice



The **gr**anule language



Modal Type Analysis



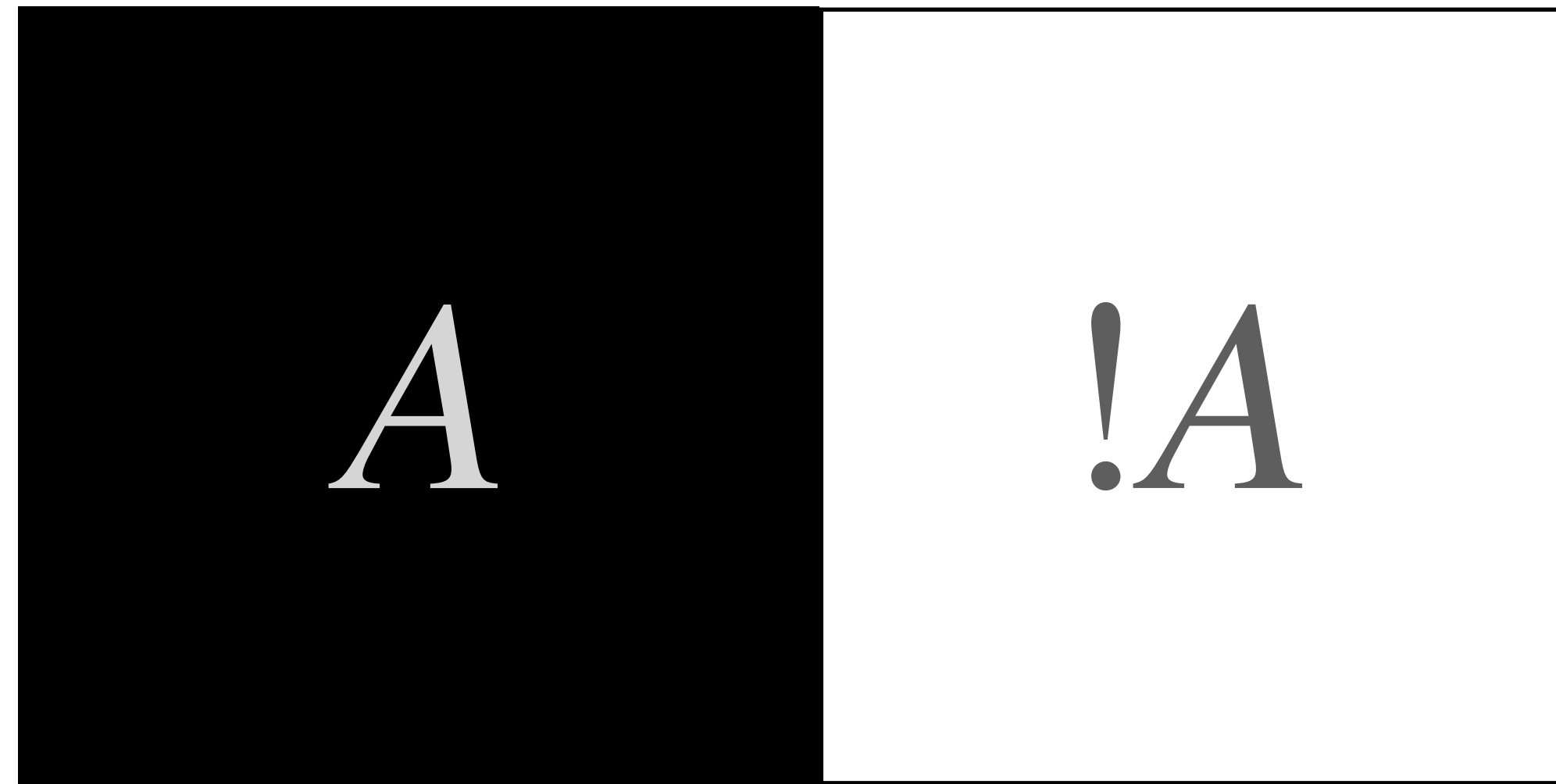
$\square, \diamond, !, ?, M$

Graded Modal Type Analysis



$\square_r, \diamond_g, !_n, \dots$

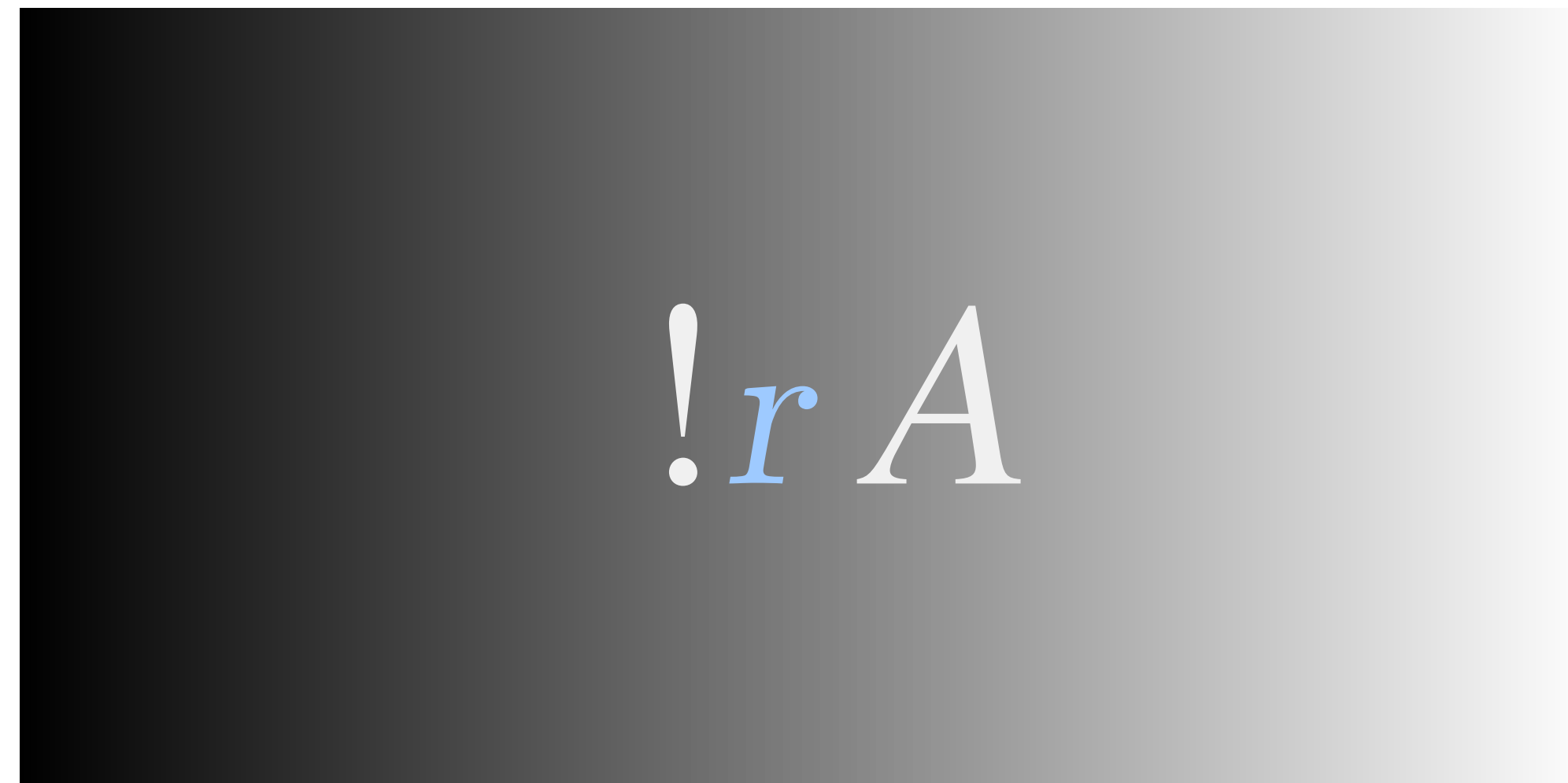
**Modal
Type
Analysis**



linear

non-linear

**Graded
Modal
Type
Analysis**



$r \in \mathcal{R}$
semiring

linear

non-linear



Quantitative Program Reasoning with Graded Modal Types

DOMINIC ORCHARD, University of Kent, UK

VILEM-BENJAMIN LIEPELT, University of Kent, UK

HARLEY EADES III, Augusta University, USA

In programming, some data acts as a resource (e.g., file handles, channels) subject to usage constraints. This poses a challenge to software correctness as most languages are agnostic to constraints on data. The approach of linear types provides a partial remedy, delineating data into resources to be used but never copied or discarded, and unconstrained values. Bounded Linear Logic provides a more fine-grained approach, quantifying non-linear use via an indexed-family of modalities. Recent work on *coeffect types* generalises this idea to *graded comonads*, providing type systems which can capture various program properties. Here, we propose the umbrella notion of *graded modal types*, encompassing coeffect types and dual notions of type-based effect reasoning via *graded monads*. In combination with linear and indexed types, we show that graded modal types provide an expressive type theory for quantitative program reasoning, advancing the reach of type systems to capture and verify a broader set of program properties. We demonstrate this approach via a type system embodied in a fully-fledged functional language called Granule, exploring various examples.

CCS Concepts: • **Theory of computation** → **Modal and temporal logics; Program specifications; Program verification**; *Linear logic; Type theory*.

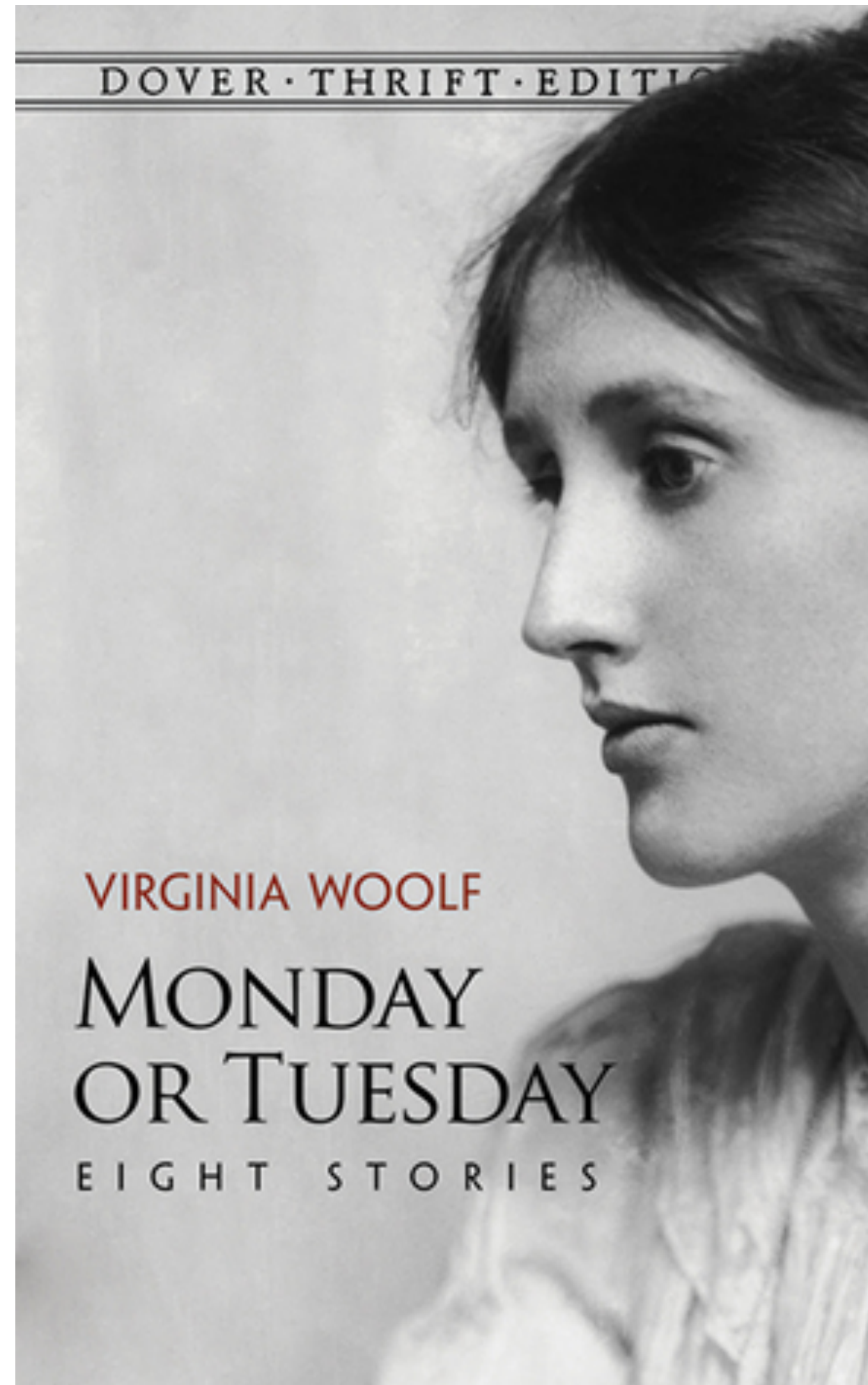
Additional Key Words and Phrases: graded modal types, linear types, coeffects, implementation

ACM Reference Format:

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning


with Graded Modal Types. Proc. ACM Program. Lang. 3, ICFP, Article 116 (August 2019). 30 pages. <https://doi.org/10.1145/332116>

Recent past





Resourceful Program Synthesis from Graded Linear Types

Jack Hughes^(✉)  and Dominic Orchard 

School of Computing, University of Kent, Canterbury, UK
{joh6,d.a.orchard}@kent.ac.uk

Abstract. Linear types provide a way to constrain programs by specifying that some values must be used exactly once. Recent work on *graded modal types* augments and refines this notion, enabling fine-grained, quantitative specification of data use in programs. The information provided by graded modal types appears to be useful for type-directed program synthesis, where these additional constraints can be used to prune the search space of candidate programs. We explore one of the major implementation challenges of a synthesis algorithm in this setting: how does the synthesis algorithm efficiently ensure that resource constraints are satisfied throughout program generation? We provide two solutions to this *resource management* problem, adapting Hodas and Miller’s input-

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. Lastly, we show how other useful structural combinators can also be automatically derived.

1 Introduction

Linear type systems capture and enforce the idea that some data travels a *linear* dataflow path through a program, being consumed exactly once. This is enforced by disallowing the structural rules of weakening and contraction on variables carrying such values. Non-linear dataflow is then typically captured by a modality !A characterising values/variables on which all structural rules are permitted [12]. This binary characterisation (linear *versus* non-linear) is made more fine-grained in Bounded Linear Logic (BLL) via



Linearity and Uniqueness: An Entente Cordiale

Daniel Marshall¹ (✉) , Michael Vollmer¹ , 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, building on recent distinct theories of linear types and uniqueness, and

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

Daniel Marshall

University of Kent, UK

dm635@kent.ac.uk

Dominic Orchard

University of Kent & University of Cambridge, UK

d.a.orchard@kent.ac.uk

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

1 Introduction

Most programming languages ascribe a notion of *type* (dynamic or static) to data, classifying *what* data we are working with—integer, string, function, etc. *Behavioural types* on the other hand capture not just *what* data is, but *how* it is calculated. One such behavioural type system for concurrency is *session types* [10], representing the behaviour of a process communicating over a channel via the channel's type.

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

Daniel Marshall

University of Kent, UK

dm635@kent.ac.uk

Dominic Orchard

University of Kent & University of Cambridge, UK

d.a.orchard@kent.ac.uk

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

1 Introduction

Most programming languages ascribe a notion of *type* (dynamic or static) to data, classifying *what* data we are working with—integer, string, function, etc. *Behavioural types* on the other hand capture not just *what* data is, but *how* it is calculated. One such behavioural type system for concurrency is *session types* [10], representing the behaviour of a process communicating over a channel via the channel's type.

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

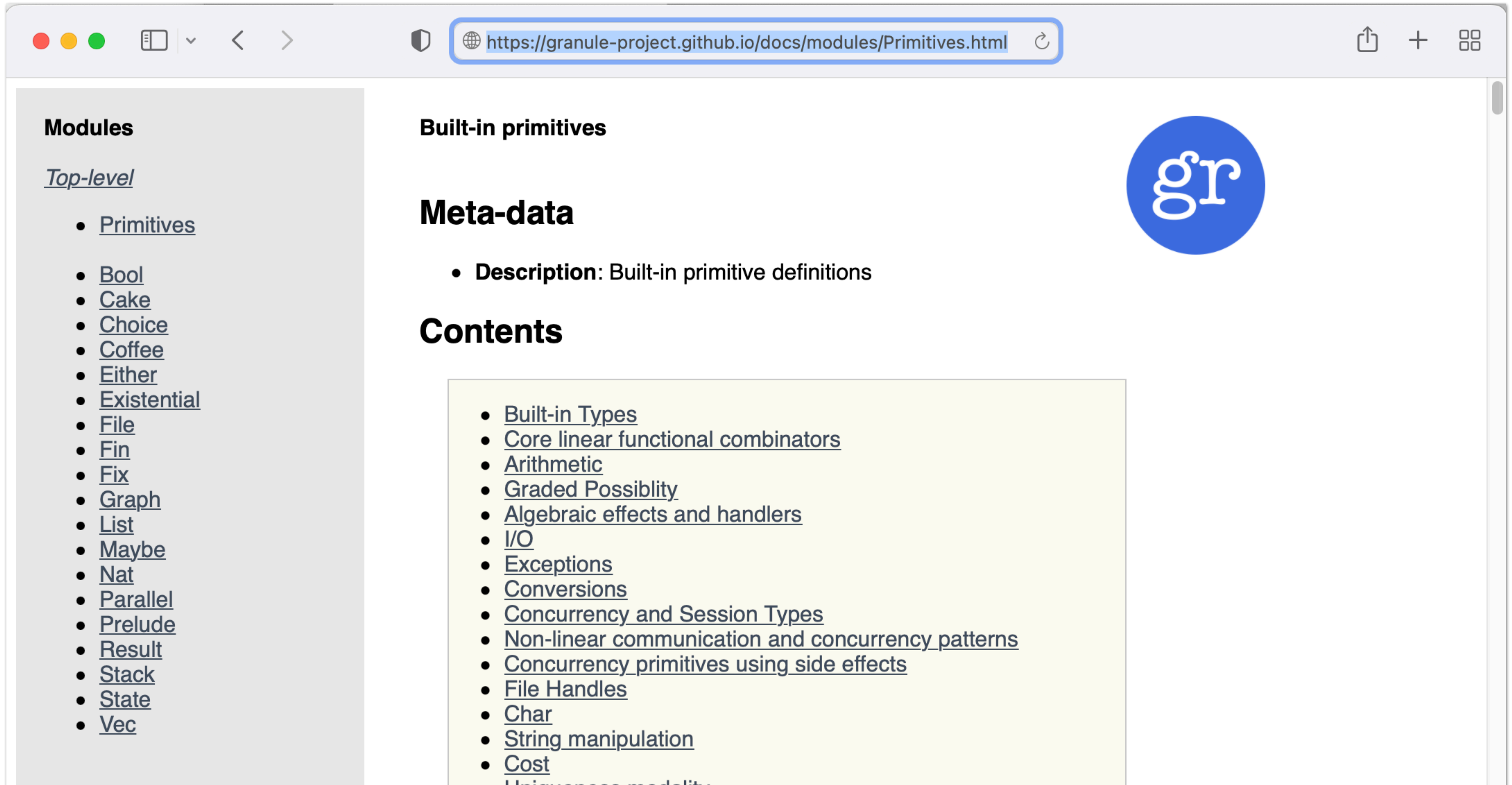
In functional programming, regular types are a subset of algebraic data types formed from products and sums with their respective units. One can view regular types as forming a commutative semiring but where the usual axioms are isomorphisms rather than equalities. In this pearl, we show that regular 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

<https://granule-project.github.io/docs>



The screenshot shows a web browser window with the address bar containing the URL `https://granule-project.github.io/docs/modules/Primitives.html`. The page content is as follows:

Modules

Top-level

- [Primitives](#)
- [Bool](#)
- [Cake](#)
- [Choice](#)
- [Coffee](#)
- [Either](#)
- [Existential](#)
- [File](#)
- [Fin](#)
- [Fix](#)
- [Graph](#)
- [List](#)
- [Maybe](#)
- [Nat](#)
- [Parallel](#)
- [Prelude](#)
- [Result](#)
- [Stack](#)
- [State](#)
- [Vec](#)


Built-in primitives

Meta-data

- **Description:** Built-in primitive definitions

Contents

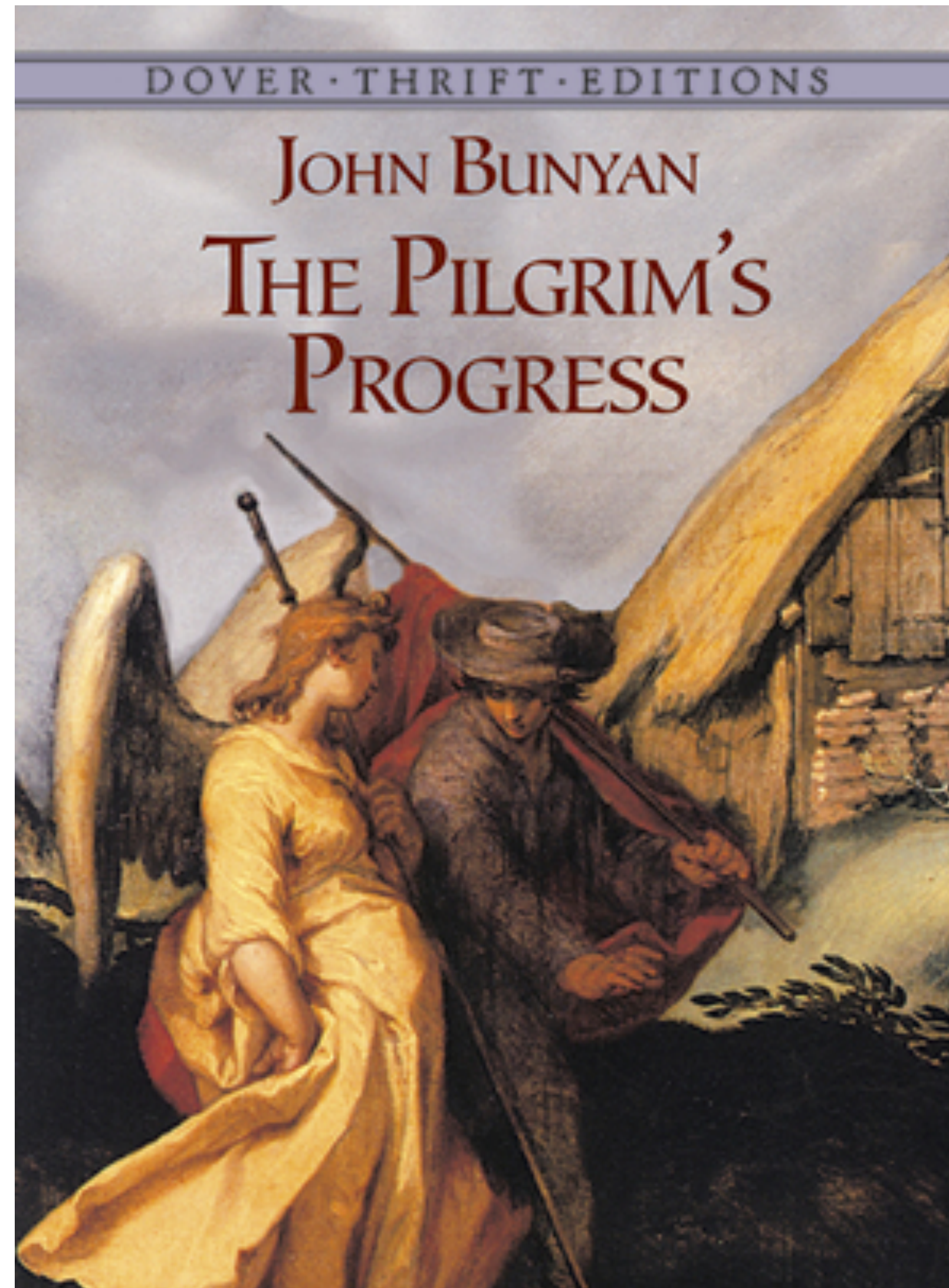
- [Built-in Types](#)
- [Core linear functional combinators](#)
- [Arithmetic](#)
- [Graded Possibility](#)
- [Algebraic effects and handlers](#)
- [I/O](#)
- [Exceptions](#)
- [Conversions](#)
- [Concurrency and Session Types](#)
- [Non-linear communication and concurrency patterns](#)
- [Concurrency primitives using side effects](#)
- [File Handles](#)
- [Char](#)
- [String manipulation](#)
- [Cost](#)
- [Uniqueness modality](#)



Some principles

- ~~No~~ Low magic
- Build things from theoretical elements
- Light syntax
- Interleave type checking and SMT
- CBV as (primary) semantics (but swappable in interpreter)

The present and ongoing



The present and ongoing

- Graded types \bowtie Algebraic effects and handlers
- Graded uniqueness (borrowing and lifetimes)
- Graded base

$\&_p A$





Graded types in Haskell (GHC 9)

```
{-# LANGUAGE LinearTypes #-}
```

```
a %r -> b
```

Graded arrow

Linear

```
a %One -> b
```

cf. linear-base:

```
a -> b
```

```
a [Lin] -> b
```

Unrestricted

```
a %Many -> b
```

```
a [Many] -> b
```

Graded modality

```
data Box r a where { Box :: a %r-> Box r a }
```




language GradedBase

$A \% r \rightarrow B$

Download and play!

<https://granule-project.github.io/>

Some more resources here from recent summer school material

<https://granule-project.github.io/splv23>

