University of Kent

Kent Academic Repository

Liepelt, Vilem, Marshall, Daniel, Orchard, Dominic, Rajani, Vineet and Vollmer, Michael (2025) *On graded coeffect types for information-flow control*. "Languages, Compilers, Analysis - From Beautiful Theory to Useful Practice Essays Dedicated to Alan Mycroft on the Occasion of His Retirement", LNCS Volume number 15500, 2025 . pp. 114-148.

Downloaded from
https://kar.kent.ac.uk/112666/ The University of Kent's Academic Repository KAR

The version of record is available from
https://doi.org/10.1007/978-3-032-08187-2_7

This document version
Author's Accepted Manuscript

DOI for this version

Licence for this version
UNSPECIFIED

Additional information

Versions of research works

Versions of Record
If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts
If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in ***Title of Journal*** , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries
If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

On Graded Coeffect Types for Information-Flow Control

Vilem-Benjamin Liepelt[1](✉), Danielle Marshall[1,3], Dominic Orchard[1,2], Vineet Rajani[1], and Michael Vollmer[1]

[1] University of Kent, Canterbury, UK
mail@vilem.net
[2] University of Cambridge, Cambridge, UK
[3] University of Glasgow, Glasgow, UK

Abstract. Graded types are an overarching paradigm that provides fine-grained reasoning by reflecting the structure of typing rules into a system of type annotations. A significant subset of graded type systems is that of *coeffect systems*, originally introduced by Petricek, Orchard, and Mycroft as a dual to effect systems, capturing the dataflow of values in a calculus by annotating variables and function types with elements of a semiring. A particularly useful instance of these graded coeffect systems is to capture security properties of data to enforce information-flow control. We examine this particular use case and give a new characterisation of a subclass of semirings which enable the key non-interference theorem of information-flow control: that less privileged observers are unable to distinguish the dependence of computations on more privileged inputs. The result relies on a logical relations proof and is mechanised in Agda. We consider its relationship to other characterisations of non-interference in the recent literature on graded types and in the historical context of coeffect and graded systems. We leverage these results for programming with security in the Granule programming language, a research language for graded types. We conclude with extensions to Granule that go beyond non-interference to *declassification*, leveraging graded types to control deliberate information leakage.

1 Introduction

An important consideration in the design of a software system is how security is handled. A typical program has access to various data sources that must be kept secret. Therefore, it is vital that such a program carefully manage data that comes from secret inputs, ensuring that the data only flows to corresponding secret outputs. A program that fails to manage this correctly can leak secret data and is said to violate *confidentiality*. To guarantee this does not happen, a system designer may choose to enforce some security policy, but how?

[3] To appear in "Languages, Compilers, Analysis - From Beautiful Theory to Useful Practice Essays Dedicated to Alan Mycroft on the Occasion of His Retirement", LNCS Volume number 15500, 2025

A well-studied method for ensuring that programs obey a specified security policy is *Information Flow Control* (IFC) [28]. IFC is a form of dependence analysis, which ensures that secret inputs do not flow to public outputs, whether this be through bug or malice. In a language-based setting, aspects of IFC can be achieved statically [1] or dynamically [28]. IFC analyses typically associate *labels* (secrecy annotations) to data, tracking these labels through a program's data and control flow.

For example, consider a function with two integer inputs that computes a pair, the first component of which is an arithmetic expression over both inputs and the second component is just the first input:

$$\texttt{process} = \lambda x.\lambda y.(x + y, x)$$

A static IFC type system might then permit the following type specification for this function, where the first input has been annotated as low security (public) and the second as high security (secret) and thus the first component of the pair is high security, and the second component is low security:

$$\texttt{process} : \texttt{Integer}^{\mathsf{Lo}} \to \texttt{Integer}^{\mathsf{Hi}} \to (\texttt{Integer}^{\mathsf{Hi}} * \texttt{Integer}^{\mathsf{Lo}})$$

Such a system would disallow a typing of the output pair that has both components marked as low security as this would constitute a potential leak of the first input. In this work we focus on using types for IFC analysis, in particular using the framework of *coeffect-and-type systems* whose notation differs to the above example, but which captures the same fundamental ideas.

Coeffect-and-type systems (hereafter *coeffect systems* for brevity) were proposed by Petricek, Orchard, and Mycroft as a dual to effect systems [52, 53], providing a general type-based analysis of context dependence in programs. Later work brought effect and coeffect systems under the umbrella of *graded type systems* [21]. In general, graded type systems provide a principled approach for analysing programs by augmenting a base type system with annotations (called *grades*) consisting of some algebraic structure which reflects aspects of the underlying system (such as semantic structure or proof structure). They can be used to reason about a variety of program properties [46], including IFC as first suggested by Petricek, Orchard, and Mycroft [52].

Most of the prior work on graded type theories for IFC is in the *effect* style (e.g., Abdadi et al.'s DCC [1]). The idea is to prevent information leaks by constraining the *outputs* that can be produced by a given program. This is often implemented using a graded monad to track the flow of information with security labels as the grades. However, there is also a less explored dual approach, which achieves IFC in the *coeffect* style via a graded comonad. The idea here is to protect information leaks by constraining the *inputs* that can be fed to a given program. In this style, grades are typically drawn from a semiring (or semiring-like structure), capturing dataflow. Prior works [21, 2, 18, 43] have proposed instantiating a semiring-graded system with a lattice of security labels to capture information-flow for the purpose of data security properties. We examine this use case in detail and propose that there is a more general class of semirings which

can be used for IFC and which enforce the crucial non-interference property: that less privileged observers cannot distinguish the dependence of a computation on more privileged inputs.

We begin with a tour through the history and development of coeffect systems, starting with the work of Alan Mycroft and his students, joining together various threads from the literature culminating in the general graded types paradigm (Section 2). From this historical narrative, we define a core graded coeffect calculus, called GrBase, based on the structural coeffect systems of Petricek, Orchard, and Mycroft [53], but reformulated within the context of the last decade of developments in graded types (Section 3). This captures the essential core of many systems in the literature discussed in Section 2. We then define a logical-relation-based semantic model for GrBase and use this to prove non-interference for a general class of pre-ordered semirings with some additional axioms which we call *non-interfering semirings* (Section 4). This serves to clarify a minimal set of constraints that need to hold of a coeffect analysis to enforce the key IFC property of non-interference. Our result is formalised in Agda.[4] The mechanisation covers the entire relational model for the graded type system and its non-interference result, but admits as postulates some basic results about substitution and strong normalisation that have been proved elsewhere in the literature for such systems.

Section 5 demonstrates an in-depth practical example of coeffect-based IFC using the functional programming language Granule,[5] where we have implemented these ideas. Section 6 concludes with a discussion of the related work of other approaches to information-flow typing and graded types. We also consider extensions to *declassification* (controlled leakage), outside of our non-interference result, and how graded types can be used to control information leakage.

2 The Road to Graded Coeffects

We define in Section 3 a core calculus with a graded type system for tracking coeffects given by a pre-ordered semiring structure. This system has arisen independently in a few forms and from a few different starting points, but, most notably for this festschrift, first due to Alan Mycroft and his students. We begin by giving a recap of the history and an overview of this space.

2.1 Coeffects as Dual to Effect Systems

In a 2013 ICALP paper *Coeffects: Unified static analysis of context-dependence*, Petricek, Orchard, and Mycroft initiated a programme of work studying a general type-based framework for analysis of *context dependence* in programs [52].

The central idea is that there is a sensible dualisation of the type-and-effect systems (hereafter *effect systems*) of Gifford and Lucassen [24, 36]. This dualisation yields a useful and flexible system for tracking program properties relating

[4] `https://github.com/granule-project/security-coeffects-mechanization`
[5] `https://github.com/granule-project/granule`

to how a program uses its context. The work builds on the close connections between effect systems and monads, fleshed out by Wadler and Thiemann [67]: for an effect judgment $\Gamma \vdash t : A, F$ describing that program t produces (at most) side effects F (where F is a member of a semilattice) then there is a translation to an equivalent judgment in a monadic meta-language $[\![\Gamma]\!] \vdash_M [\![t]\!] : M_F[\![A]\!]$ where the monadic type constructor M is annotated by the effect information F. The following two rules capture the composition of side effectful computations whilst concordantly composing the effect information in types:

$$\frac{\Gamma \vdash_M t : A}{\Gamma \vdash_M \mathsf{return}\ t : M_{\bot} A}\text{RET} \qquad \frac{\Gamma \vdash_M t_1 : M_F A \qquad \Gamma, x : A \vdash_M t_2 : M_G B}{\Gamma \vdash_M \mathsf{do}\ x \leftarrow t_1; t_2 : M_{F \vee G} B}\text{BIND}$$

Later work formalises this "annotated" monad structure M_F as the more general notion of *graded monads* [33, 50]. (Mycroft et al. later generalised this further, considering in depth the paradigm of capturing effects via grading [44]).

From this starting point of the relationship between effects and monads, the ICALP 2013 paper sought to find the corresponding annotated type system for *comonads*, which were gaining in popularity for structuring notions of context dependence in semantics [62, 14], computational logic [12, 11, 7] (by former PhD students of Alan), and programming [60, 61, 49, 48], amongst others.

The result of that paper is a general type-and-coeffect system (hereafter *coeffect system*) with judgments of the form [52]:

$$C^S \Gamma \vdash t : A$$

capturing that for a program t of type A, alongside the usual free variables Γ, the program also requires (at least) the 'context' S (with C merely representing a syntactical anchor for the comonadic denotational model) where S is drawn from a lattice $\mathcal{S}$ with additional monoid structure $(\otimes, 1)$.[6] Key typing rules:

$$\frac{x : A \in \Gamma}{C^1 \Gamma \vdash x : A}\text{VAR} \qquad \frac{C^{R \wedge S}(\Gamma, x : A) \vdash t : B}{C^R \Gamma \vdash \lambda x.t : C^S A \to B}\text{ABS}$$

$$\frac{C^R \Gamma \vdash t_1 : C^S A \to B \quad C^T \Gamma \vdash t_2 : A}{C^{R \vee (S \otimes T)} \Gamma \vdash t_1\ t_2 : B}\text{APP}$$

The (VAR) rule describes that using a variable amounts to some usage of the context denoted by the $1 \in \mathcal{S}$ element. Reading top-down, the (ABS) rule says that the context-dependence of a function's body can be split between the 'call site' by capturing the dependence S on the function arrow (written $C^S A \to B$) leaving the remaining requirements R to be made at the 'definition site'. Reading bottom-up, the (ABS) says that the context *available* to a function is a combination of the current context R and the context provided by the function argument S. The (APP) rule then shows how these requirements / capabilities

[6] The ICALP 2013 paper actually uses $\oplus$ and e as its syntax for the monoidal part of the coeffect algebra structure, but thinking of this monoid in multiplicative terms fits better with the literature that came after.

come together when applying a function, using the monoidal $\otimes$ to combine the context T of the argument t_2 with the requirements S of the function t_1.

The paper gives various examples, including one using sets of labels to capture resources required from the context, e.g., hardware devices or global variables, with $\mathcal{S} = \mathcal{P}(\mathit{resources})$. The usual lattice of sets and monoid $\otimes = \cup$ and $1 = \emptyset$ is used with a primitive for accessing resources $C^{\{?\mathsf{a}\}}\Gamma \vdash ?\mathsf{a} : \rho$. For example:

$$C^{\{?\mathsf{gps},?\mathsf{compass}\}} \ldots \vdash (\lambda c.\, \mathit{locate}\ ?\mathsf{gps}\ c)\ ?\mathsf{compass} : \ldots$$

An interactive web-based essay by Tomas Petricek (Alan's student at the time) provides a playground exploring this example and others,[7] and Petricek's PhD thesis discusses 'systems capabilities as coeffects' in more detail [51].

This line of work has been picked up by several practical programming languages. One example of this is Hack, via its *contexts and capabilities* [59].

The paper then develops a categorical semantics which introduces the notion of *indexed comonads* which, in modern terminology, are known as *graded comonads* [21], dualising graded monads but with additional structure for managing the more complex structure of contexts.

A key aspect of the paper is that the system is general, parameterised by a coeffect algebra $\mathcal{S}$ which can then be instantiated to various interesting and useful analyses. The paper includes an example based on whether the entire free variable context Γ is used (live) or is unused (dead). The paper concludes by pointing out that it would be more useful in that case to track liveness of variables (whether they are 'relevant') on a *per variable* basis rather than *per context* and suggests the notion of a 'structural' coeffect system where coeffect annotations are formed out of finite products matching the structure of the context, e.g., with abstraction rule:

$$\frac{C^{R\times S}(\Gamma, x : A) \vdash t : B}{C^{R}\Gamma \vdash \lambda x.t : C^{S}A \to B}\,\mathrm{ABS}'$$

These structural coeffects (which the present paper builds on in Section 3) then lend themselves to analysing how variables are used, separately to wider notions of context dependence which were the main focus of the ICALP 2013 paper. In this teasing of the more fine-grained structural coeffect system, the authors suggested that such a system could be used for tracking secure information flow along the lines of DCC [1], the topic of the present paper.

A follow-up paper at ICFP 2014, titled *Coeffects: A calculus of context-dependent computation* also by Petricek, Orchard, and Mycroft, builds a unified system in which both the ICALP 2013 'flat' (whole context) coeffects and structural (per variable) coeffects can be expressed as instances of a single system via a coeffect structure akin to a (shape-indexed) module over a semiring. We do not replay the full generality of the system here but briefly overview a subset specialised to the subclass of coeffect structures which encode structural coeffects. In this approach judgments have the form:

$$\Gamma @ R \vdash t : A$$

[7] `https://tomasp.net/coeffects/`

where R is a vector $\langle r_1, \ldots, r_n \rangle$ of coeffect annotations $r_i \in \mathcal{R}$ drawn from a pre-ordered semiring[8] where $n = |\Gamma|$, i.e., the structure and shape of R matches the context. Selected key typing rules are then:

$$\frac{}{x : A @ \langle 1 \rangle \vdash x : A}\text{VAR}$$

$$\frac{\Gamma, x : A @ R \times \langle r \rangle \vdash t : B}{\Gamma @ R \vdash \lambda x.t : A \xrightarrow{r} B}\text{ABS} \qquad \frac{\Gamma_1 @ R \vdash t_1 : A \xrightarrow{r} B \quad \Gamma_2 @ S \vdash t_2 : A}{\Gamma_1, \Gamma_2 @ R \times (r * S) \vdash t_1\, t_2 : B}\text{APP}$$

$$\frac{\Gamma_1, y : B, z : B, \Gamma_2 @ R \times \langle s, t \rangle \times Q \vdash t : A}{\Gamma_1, x : B, \Gamma_2 @ R \times \langle s + t \rangle \times Q \vdash [x/z][x/y]t : A}\text{CONTR} \qquad \frac{\Gamma @ R \vdash t : A}{\Gamma, x : B @ R \times \langle 0 \rangle \vdash t : A}\text{WEAK}$$

The (VAR) rule now annotates a singleton context with a singleton vector using the multiplicative unit $1 \in \mathcal{R}$ of the semiring to denote usage. The (ABS) splits off the coeffect for the variable being bound, where the syntax is now $A \xrightarrow{r} B$ for a function which uses its input according to r (rather than $C^r A \to B$ as in the ICALP 2013 paper). The (APP) rule composes two disjoint contexts but scales the requirements of S for the argument by the coeffect of the function r, where the semiring's multiplication is lifted to a scalar multiplication of a coeffect vector by a coeffect. Lastly, the (CONTR) and (WEAK) rules provide contraction and weakening using the additive part of the semiring. A somewhat similar approach will be used and explained in more detail in the present paper.

2.2 Coeffects as a generalisation of Bounded Linear Logic

Independently to Petricek, Orchard, and Mycroft's ICALP 2013 paper, two very similar systems arose independently of each other in ESOP 2014 (by Ghica and Smith [23] and Brunel et al. [15]), coming from a different angle: generalising *bounded linear logic* (BLL), which is itself is a generalisation of linear logic.

Linear logic is a logic of *resources* in which propositions must be used exactly once, i.e., they cannot be discarded and cannot be copied [25]. A corresponding system of linear types can then be used to control data as a resource [66]; for example, to enforce stateful protocols, e.g., that a file handle is opened, read from and written to, then closed once and never accessed again after [68]. Non-linearity is provided by the !-modality (pronounced 'of course' or 'bang') which gives a binary view, delineating linear and non-linear propositions or values. Bounded Linear Logic (BLL) refines the binary view, replacing ! with a *family* of indexed modal operators $!_r$ where the index r provides an upper bound on usage [26], e.g., $!_4 A$ represents a value A which may be used up to 4 times.

[8] In fact, the ICFP 2014 paper refers to this as the *coeffect scalar* structure $\mathcal{R}$ which comprises a pair of monoids with distributive laws between them but does not mention the need for 0 (called ign in ICFP 2014) to be absorbing or that + is commutative, however the paper points to the proofs in the thesis of Orchard which explains additional axioms which are required of the coeffect scalar in order for subject reduction/type preservation to hold [47] albeit on a slightly different presentation of the structure. The modern rendering in terms of semiring is more clear.

Dal Lago and Gaboardi applied linear typing to PCF with BLL-style modalities indexed by usage bounds which could depend on type-level natural number indices [34, 20]. Observing that the natural number terms form a semiring and the system can be usefully generalised to arbitrary semirings, this approach was generalised by Brunel, Gaboardi, Mazza, and Zdancewic in an ESOP 2014 paper *A Core Quantitative Coeffect Calculus* [15]. The work was developed independently of the work on coeffects, but the authors discovered the terminology of coeffects towards the end of their project[9] and so helpfully worked that terminology into their description of the system.

The core system has typing contexts which can contain both linear variables and non-linear variables, which are marked with an element of a semiring. A dual-calculus style presentation is possible, splitting the contexts, though the ESOP 2014 work combined these into a single context syntax. A family of modal operators $\Box_r A$ where r is drawn from the semiring is then used to internalise the coeffect information, with the following subset of its typing rules:

$$\frac{}{x : A \vdash x : A}\text{var} \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\text{abs} \quad \frac{\Gamma \vdash t_1 : A \multimap B \quad \Delta \vdash t_2 : A}{\Gamma + \Delta \vdash t_1\ t_2 : B}\text{app}$$

$$\frac{\Gamma, x : A \vdash x : A}{\Gamma, x : [A]_1 \vdash x : A}\text{der} \quad \frac{\Gamma \vdash t : A}{r * \Gamma \vdash [t] : \Box_r A}\text{pr}$$

The variable and abstraction rules are standard for linear type systems. The application rule is also standard but includes a context addition operation that is undefined if the contexts are not disjoint in their linear assumptions and which adds together (using the semiring addition) the coeffects of any non-linear variable assumptions that appear in both contexts. The 'dereliction' rule (der) links linearity with coeffects, marking a linear assumption with 1 of the semiring. The 'promotion' rule (pr) then introduces a coeffect-graded modality at grade r, requiring that the inputs are also scaled by r (and implicitly that the context contains no linear variables).

Independently to the work of Brunel et al., another paper appeared at the very same ESOP (2014) proposing essentially the same idea: Ghica and Smith provided a generalisation of BLL in their paper *Bounded Linear Types in a Resource Semiring* [23]. Whilst they explicitly generalise BLL, the calculus presented does not provide a base notion of linearity. Instead, similarly to structural coeffects, every assumption is associated with an element of a semiring and the function type always has a semiring element associated with its input (in their case written as $(J.A) \multimap B$ for semiring element J, calling back to the linear logic roots at least via the appearance of $\multimap$). The resulting system has very similar expressivity and power to that of Brunel et al., with 1 of the semiring denoting usage, 0 denoting discarding, multiplication for composition and addition for contraction. Ghica and Smith give a particularly interesting use case for coeffects for capturing 'hardware schedules' in circuit synthesis.

[9] Personal communication.

2.3 The Graded Types Paradigm and Further Systems

In summary, the notion of coeffects as a semiring (or semiring-like) type-based analysis of how a program depends on its variables arose independently as both a generalisation of effects (ICALP 2013, 2014) and a generalisation of bounded linear logic (ESOP 2014, two independent papers).

At ICFP 2016, Gaboardi et al. went on to generalise the ESOP 2014 paper of Brunel et al., using now the terminology of *grading*, providing a linear logic with a *graded comonadic* modality (essentially like the system of Brunel et al.) but also adding graded monads for tracking effects and (graded) distributive laws for capturing effect-coeffect interaction [21]. One of the examples given in their work is a graded comonad with a two-point lattice of 'high' and 'low' security markers for tracking security as a coeffect. This is the first such instance actually showing the details of using coeffects for security, following the cursory mention by Petricek, Orchard, and Mycroft in ICALP 2013. However, the example was not developed to the point of proving non-interference, though the categorical semantics captures the idea implicitly.

Following this calculus with graded monads and comonads, Orchard et al. presented an implementation of the Gaboardi et al.'s work as a functional programming language, called Granule, extended with polymorphism and GADTs [46]. The language has served as a research vehicle for exploring the applications of graded type systems, including for example program synthesis [30, 32, 31] and capturing uniqueness and borrowing for mutable structures [40, 38]. The paper proposes the general paradigm of *graded modal types*, encompassing both graded comonadic types for coeffects, graded monadic types for effects, and suggesting there may be other flavours of graded modality. The general paradigm is that

> "graded modal types carry information about the semantic structure of programs, and along with a suitably expressive type system, provide a mechanism for specifying and verifying properties not captured by existing type systems" [46].

Various other works have since proposed related general systems or have deployed such systems. For example, Abel and Bernardy give a general characterisation of semiring-graded type systems (similar to structural coffects but also with an added graded modality) and study in detail its semantic models [2]. They consider information-flow control as a key application and prove a non-interference result, which we discuss in this paper in more detail in Section 4.

As a notable application of graded types, GHC Haskell now provides linear types as an extension, the implementation of which uses a structural coeffect system, providing a way to describe linear typing via a $\{0, 1, \omega\}$ ("none, one, tons") semiring [8].

Lastly, coeffect-like grading has also been a subject of much study in the context of dependent types. Notably, McBride proposed a dependent type theory with coeffects for tracking whether values are used or not as a way of reconciling linear and dependent types [41]. The system was refined by Atkey under the name Quantitative Type Theory (QTT) [6], which has then become the basis

for the Idris 2 implementation [13]. Several other dependent type systems have then included graded types for capturing coeffects, including Grad [18], Graded Modal Dependent Type Theory (GrTT) [43], and a similar fully formalised version by Abel et al. with universes and erasure properties [3].

Our focus here is on simple and polymorphic type systems, rather than dependent type theories; applying the ideas here to a more powerful dependent type theory is further work.

2.4 Linear Base versus Graded Base

As seen, there has been a proliferation in graded type systems, particularly focusing on coeffects, in the last decade. A common theme is the use of a (preordered) semiring to structure the coeffect analysis. There are two dominant styles of system that have emerged in this vein: those which take linear logic as a basis and those which do not. In the former case ("linear base"), contexts comprise a mixture of linear assumptions and graded assumptions, function types are linear functions $\multimap$, and a graded modality $\Box_r$ internalises the grading from the context. Such systems include the ESOP 2014 system of Brunel et al. [15], the ICFP 2016 system of Gaboardi et al. [21], and Granule [46]. In the latter case ("graded base"), coeffect information is pervasive: every typing assumption has a grade, function types are annotated with a grade describing the input use, and the inclusion of a graded modality is optional (though useful for flexibility). Such systems include the original coeffect systems of Petricek, Orchard, and Mycroft [52, 47], the ESOP 2014 system of Ghica and Smith [23], the general system of Abel and Bernardy [2], and the dependent type systems mentioned above. Recent theoretical work by Vollmer et al. has illustrated the relationship between these two styles more closely, by showing that there is a mutual embedding between the two approaches in their system [63].

The main differences between the types and contexts of linear vs. graded base systems can be summarised as follows:

$A ::= \Box_r A \mid A \multimap B$	$A ::= \Box_r A \mid A^r \to B$
$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]_r$	$\Gamma ::= \emptyset \mid \Gamma, x :_r A$
(linear base)	(graded base)

Linear base has two kinds of assumptions (linear and graded) and a linear arrow; graded base has one kind of assumption (graded) with a graded function arrow.

It has recently been shown that both styles are equally expressive, i.e. any linear base term can be emulated in graded base and vice versa [35]. In this work, we continue in the graded base style, used in the early coeffect systems, using this as our basis for a core calculus in which we can then study IFC via coeffects.

3 A Core Graded Coeffect Calculus

Following the approaches described in the previous section, we focus on GrBase, a minimal core calculus in the style of Petricek, Orchard, and Mycroft's structural coeffect system [53], but using the language and terminology of grades. We thus describe our system as having *graded coeffects* structured by a *pre-ordered semiring*: grades are associated to every free variable in the context and track how it is used, with a graded function type to describe how the input is used. Thus there is no base notion of linearity as found in systems building directly from the linear logic tradition (i.e., [15, 21]), but we do include explicit graded modalities for internalising grades into types separately to functions as this allows more flexibility. As discussed in the previous section this style of core calculus is becoming routine and appears in various works now in slightly different forms [53, 41, 8, 18, 23, 2, 69].

We supplement the presentation of GrBase with real examples written in the Granule programming language. (Real, in the sense that they are embedded in the source of this paper such that they can be type checked by the Granule system.) As mentioned in Section 2.4, Granule, by default, represents a linear base system. However, it can be run in graded base mode via the `language GradedBase` pragma, which we are doing for this paper, as this makes Granule a strict superset of GrBase. Granule features a rich pattern language and equational-style function definitions with syntax similar to Haskell's.

3.1 Syntax and Typing

The syntax of GrBase is essentially that of the λ-calculus extended with a graded necessity modality and products (along with their unit), and booleans (for control flow). The syntax of types is:

$$A ::= \; A \otimes B \mid \mathsf{Unit} \mid \mathbb{B} \mid A^{r} \to B \mid \Box_{r} A \qquad (\textit{types})$$

Types comprise products, units, booleans, graded functions $A^{r} \to B$, and a graded modality $\Box_{r} A$. The calculus is parameterised by a pre-ordered semiring $(\mathcal{R}, \cdot, 1, +, 0, \leq)$ where $\cdot$ and $+$ are monotonic with respect to the pre-order $\leq$. For function types, a grade $r \in \mathcal{R}$ is associated with the input type, describing the usage of the input within the inhabiting function. The graded modality $\Box_{r} A$ captures a *capability* to use a value typed A according to grade $r \in \mathcal{R}$.

The syntax of GrBase is as follows (explained later with reference to typing):

$$\begin{aligned} t ::= \; & \underbrace{x \mid t_1\ t_2 \mid \lambda x.t}_{\lambda\text{-calculus}} \mid \underbrace{[t] \mid \mathsf{let}\,[x] = t_1\ \mathsf{in}\ t_2}_{\text{graded modality}} \\ & \mid \underbrace{(t_1, t_2) \mid \mathsf{let}\ (x, y) = t_1\ \mathsf{in}\ t_2}_{\text{products}} \mid \underbrace{() \mid \mathsf{let}\ () = t_1\ \mathsf{in}\ t_2}_{\text{unit}} \\ & \mid \underbrace{\mathtt{ff} \mid \mathtt{tt} \mid \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3}_{\text{booleans}} \end{aligned} \qquad (\textit{terms})$$

$$\frac{}{0 \cdot \Gamma, x :_1 A \vdash x : A}\ \textsc{TyVar} \qquad \frac{\Gamma, x :_r A \vdash t : B}{\Gamma \vdash \lambda x.t : A^r \to B}\ \textsc{TyAbs}$$

$$\frac{\Gamma_1 \vdash t_1 : A^r \to B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + r \cdot \Gamma_2 \vdash t_1\ t_2 : B}\ \textsc{TyApp} \qquad \frac{\Gamma, x :_r A, \Gamma' \vdash t : B \qquad s \leq r}{\Gamma, x :_s A, \Gamma' \vdash t : B}\ \textsc{TyApprox}$$

$$\frac{\Gamma \vdash t : A}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{TyPr} \qquad \frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x :_r A \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,[x] = t_1\,\mathsf{in}\,t_2 : B}\ \textsc{TyBoxElim}$$

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \textsc{TyPairIntro}$$

$$\frac{\Gamma_1 \vdash t_1 : A_1 \otimes A_2 \qquad \Gamma_2, x : A_1, y : A_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ (x, y) = t_1\ \mathsf{in}\ t_2 : B}\ \textsc{TyPairElim}$$

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{Unit}}\ \textsc{TyUnitIntro} \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{Unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ () = t_1\ \mathsf{in}\ t_2 : B}\ \textsc{TyUnitElim}$$

$$\frac{}{0 \cdot \Gamma \vdash \mathtt{tt} : \mathbb{B}}\ \textsc{TyTrue} \qquad \frac{}{0 \cdot \Gamma \vdash \mathtt{ff} : \mathbb{B}}\ \textsc{TyFalse}$$

$$\frac{\Gamma_1 \vdash t_1 : \mathbb{B} \qquad \Gamma_2 \vdash t_2 : A \qquad \Gamma_2 \vdash t_3 : A \qquad r \leq 1}{(r \cdot \Gamma_1) + \Gamma_2 \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : A}\ \textsc{TyIfThenElse}$$

Fig. 1: Typing rules for GrBase. Regarding the direction of the ordering in Approx, note Remark 1.

Typing is via judgments $\Gamma \vdash t : A$, assigning a type A to term t under the context Γ. Every free variable type assumption in Γ has an associated semiring element called its *grade*:

$$\Gamma ::= \emptyset \mid \Gamma, x :_r A$$

The order of assumptions in contexts is not relevant, i.e., contexts can be arbitrarily reordered.

Figure 1 provides the typing rules. The Var rule accounts for the use of a particular variable, annotated via the grade $1 \in \mathcal{R}$ meaning it has been used. The remaining variables in the context are all graded 0, provided by scaling Γ by 0 and the semiring property that 0 is absorbing. *Scalar multiplication* is defined:

Definition 1 (Scalar context multiplication). *For all r, Γ then $r \cdot \Gamma$ is defined by induction on Γ:*

$$r \cdot \emptyset = \emptyset$$
$$r \cdot (\Gamma, x :_s A) = (r \cdot \Gamma), x :_{r \cdot s} A$$

In the conclusion of the application rule we also see the use of scalar context multiplication where $r \cdot \Gamma_2$ scales the grades of Γ_2 by r to account for the usage of the argument t_2 by the function t_1.

The Abs rule is standard modulo capturing of the assumption's grade r onto the function arrow (the *latent coeffect* [53]). The App rule shows a key principle of combining the typing contexts for subterms using the *context addition* operation $\Gamma + \Gamma'$ which is used in the typing rules any time the contexts of two subterms need to be combined, applying semiring addition on the grades of variables occurring in each context:

Definition 2 (Pointwise context addition). *For all Γ_1, Γ_2 of the same shape (where the shape of a context is the list of variable-type assumptions),* context addition *is defined recursively as:*

$$\emptyset + \emptyset = \emptyset$$
$$(\Gamma, x :_r A) + (\Gamma', x :_s A) = (\Gamma + \Gamma'), x :_{r+s} A$$

Example 1. A classic example is the semiring of natural numbers $\mathbb{N}$ with the multiplicative and additive parts provided by multiplication and addition on naturals. Taking the ordering to be equality (i.e. $\leq$ is defined as $\equiv$), we obtain an analysis of exact usage, meaning that, the graded modalities 'count' how many times, exactly, a variable is referenced.

For example, the following typing derivation fragment shows the role of multiplicative scaling to account for how many times a variable is used:

$$\dfrac{\dfrac{\emptyset \vdash (\lambda y.(y, y)) : (A \otimes A)^2 \to (A \otimes A) \otimes (A \otimes A) \qquad x :_2 A \vdash (x, x) : A \otimes A}{x :_4 A \vdash (\lambda y.(y, y))\,(x, x) : (A \otimes A) \otimes (A \otimes A)}\ \textsc{App}}{\emptyset \vdash \lambda x.(\lambda y.(y, y))\,(x, x) : A^4 \to (A \otimes A) \otimes (A \otimes A)}\ \textsc{Abs}$$

Thus the type of the innermost function $\lambda y.(y, y)$ captures that its input its used twice; the type of (x, x) captures that x is used twice; the application $(\lambda y.(y, y))\,(x, x)$ therefore is typed with x graded at 4 due to multiplication.

In Granule, graded function types $A^r \to B$ are written as `A % r → B`, where `r` is of kind coeffect. This syntax is reminiscent of Linear Haskell and becomes available in Granule through the `language GradedBase` pragma.

We capture an example in Granule similar to the above (but β-reduced) as the following polymorphic function:

```
quad : ∀ {a : Type}. a % 4 → ((a, a), (a, a))
quad x = ((x, x), (x, x))          -- o.k.
```

The next (non)example has an incorrect type, showing that for discrete naturals, we cannot over-approximate the usage:

```
quad_bad : ∀ {a : Type}. a % 5 → ((a, a), (a, a))
quad_bad x = ((x, x), (x, x))      -- type error!
```

Graded modalities are introduced by 'promotion' (TyPr), scaling graded assumptions in Γ via scalar multiplication. Graded modalities are eliminated by $\mathsf{let}\,[x] = t_1\ \mathsf{in}\ t_2$ connecting the graded modal box at r in the type of t_1 with an assumption of grade r in the free variable context of t_2.

Example 2. The grade on function types is rather coarse grained: it summarises how the *entire* input is used. In the case of a compound type, e.g. products, we may want more fine-grained information about how each subpart is used, for which we can use the graded modality. For example, to define first projection on pairs we use the graded modality at grade 0 for the unused second component:

$$\dfrac{\dfrac{\dfrac{}{z :_1 A \otimes \Box_0 B \vdash z : A \otimes \Box_0 B}\,\textsc{Var} \quad \dfrac{\dfrac{}{y :_1 \Box_0 B \vdash y : \Box_0 B}\,\textsc{Var} \quad \dfrac{}{x :_1 A, y' :_0 B \vdash x : A}\,\textsc{Var}}{x :_1 A, y :_1 \Box_0 B \vdash \mathsf{let}\,[y'] = y\,\mathsf{in}\,x : A}\,\textsc{BoxE}}{z :_1 A \otimes \Box_0 B \vdash \mathsf{let}\,(x, y) = z\,\mathsf{in}\,\mathsf{let}\,[y'] = y\,\mathsf{in}\,x : A}\,\textsc{PairE}}{\emptyset \vdash \lambda z.\mathsf{let}\,(x, y) = z\,\mathsf{in}\,\mathsf{let}\,[y'] = y\,\mathsf{in}\,x : (A \otimes \Box_0 B)^1 \to A}\,\textsc{Abs}$$

Thus, we first eliminate the incoming product value z into its components x and y which both get used once. The component y however is graded modal, of type $\Box_0 B$, and is eliminated to a graded assumption $y' :_0 B$ which can thus be discarded due to the 0 grade.

In Granule, sticking to the constructs supported by GrBase, we would write the following equivalent function:

```
fst' : ∀ {a b : Type} . (a, b [0]) % 1 → a
fst' = \z → let (x, y) = z in let [y'] = y in x
```

Granule provides additional syntactic sugar whereby elimination of graded modalities can be folded into pattern matching. Furthermore, we can elide the grade on an arrow if it is 1. A more idiomatic Granule equivalent to the above is thus:

```
fst : ∀ {a b : Type}. (a, b [0]) → a
fst (a, [b]) = a
```

In the case of **if-then-else**, we introduce an additional grade r to scale the incoming context of t_1 with the requirement that the grade r must approximate 1 (written $r \leq 1$) as computing a conditional incurs a usage to inspect the boolean. Once control flow is in the picture, *approximation* becomes important.

Example 3. A useful semiring is that of *intervals* over another semiring, e.g., the semiring of natural numbers intervals, formed by pairs $\mathbb{N} \times \mathbb{N}$, written here as $r..s$ in which s is always greater than or equal r under the normal ordering for $\mathbb{N}$. The semiring operations are then those of interval arithmetic, with ordering:

$$r_1..r_2 \leq s_1..s_2 = r_1 \leq_{\mathbb{N}} s_1 \wedge s_2 \leq_{\mathbb{N}} r_2$$

Thus the interval $r_1..r_2$ approximates $s_1..s_2$ if it is larger in conventional natural number terms (Remark 1 discusses the direction of the ordering and why it looks opposite to how we expect it for numbers). The following shows an example of the typing for conditionals using this semiring and approximation (eliding the typing of the 'else' branch):

$$\dfrac{\dfrac{}{x :_{1..1} \mathbb{B} \vdash x : \mathbb{B}}\,\textsc{Var} \quad \dfrac{\dfrac{}{y :_{1..1} A, z :_{0..0} A \vdash y : A}\,\textsc{Var}}{y :_{0..1} A, z :_{0..1} A \vdash y : A}\,\textsc{Approx} \quad r..r \leq 1..1}{x :_{r..r} \mathbb{B}, y :_{0..1} A, z :_{0..1} A \vdash \mathsf{if}\,x\,\mathsf{then}\,y\,\mathsf{else}\,z : A}\,\textsc{If}$$

Approximation means that we can approximate the grades for both x and y to be $0..1$, capturing that they are used either 0 or 1 times in this computation.

The typing of the conditional requires that $r..r \leq 1..1$ (since $1 = 1..1$ in this semiring) and thus we ensure that the usage of x, to decide the conditional, is recorded faithfully.

Granule provides a range of builtin semirings, including:

- `Nat` Discrete naturals, denoting exact usage.
- `Sec` Security levels `Lo` and `Hi`, similar to the lattice used in this paper (defined soon in Definition 3).
- `Q` Rationals for sensitivity analyses (somewhat emulating Gaboardi et al.'s system for differential privacy [20]).

Semirings can also be built from other data or combined using the following higher-order operations:

- `Set -` Sets of user-defined labels, where $* = \cup$ and $+ = \cap$.
- `SetOp -` Sets of user-defined labels, where $* = \cap$ and $+ = \cup$.
- `Interval -` Lower and upper bounds on usage.
- `- × -` Product of two semirings, e.g. one to three usages at level `Hi` is provided by a product `(1..3, Hi) : Interval Nat × Sec`.

`Set` and `SetOp` present a set-based lattice coeffect over user-defined labels. Intervals can be used to give lower and upper bounds; and products of coeffects allow the combination of coeffects via pointwise products.

Granule supports grade polymorphism, but also polymorphism on the semiring itself, allowing general definitions that can be instantiated at arbitrary semirings. Granule introduced semiring-polymorphism, which is appearing now in other, similar systems; Bianchini et al. adapt graded coeffects to an Object-Oriented setting, supporting different semirings within a single system and user-defined semirings via inheritance [9, 10]. In Granule, new semirings can currently only be defined by extending the implementation. Exposing an interface for user-defined semirings, akin to Bianchini et al., is further work.

Granule generalises GrBase to include sum types and arbitrary, user-defined algebraic data types. We present GrBase as a simple core here, with just booleans for control flow to focus on the essential details for the model and non-interference result of Section 4.

Since lattices are semirings, we can consider instantiating GrBase with lattices, where the elements of the lattice provide some kind of labeling on data. Such a semiring has been proposed as a way to capture IFC, with the grades as security levels [21, 2, 18, 43]. We focus on a 2-point lattice for now:

Definition 3 (2-point semiring of security levels [21]). *Let $\mathcal{L} := \{H, L\}$ be a set of abstract labels, denoting 'high' and 'low' security permissions respectively, with a lattice formed by the total order with $L \leq H$. This lattice is a semiring with multiplicative structure $\cdot := \vee$ and $1 := L$ and additive $+ := \wedge$ and $0 := H$.*

The additive part means contraction serves to gives us the most permissive label (the greatest-lower bound) to avoid passing secrets to the public domain, e.g.,:

$$\frac{\Gamma, x : [A]_{\mathsf{H}} \vdash t : B \qquad \Gamma', x : [A]_{\mathsf{L}} \vdash t' : B'}{(\Gamma + \Gamma'), x : [A]_{\mathsf{L}} \vdash (t, t') : B \otimes B'} \tag{1}$$

since $\mathsf{H} + \mathsf{L} = \mathsf{H} \wedge \mathsf{L} = \mathsf{L}$, i.e., x must be labelled as L in the concluding term since the subterm t' uses x in a low-security way.

We argue in Section 4 that a lattice is more general than what is needed to establish non-interference; instead, there is an intermediate point between semirings and lattices which provides enough structure to guarantee non-interference.

Remark 1 (On the direction of the ordering). Note that since we have $1 = \mathsf{L}$, $0 = \mathsf{H}$ and $\mathsf{L} \leq \mathsf{H}$, we have $1 \leq 0$. While this is unusual vis à vis the coeffect literature [46], the ordering turns out to be an arbitrary choice. In this work we chose the option that is common in the security literature (see e.g. [1]), as it avoids having 'high less than low' ($\mathsf{H} \leq \mathsf{L}$).

By the ordering and the semiring structure, high-security inputs H cannot flow to low security outputs L and thus there are no typing derivations for:

$$\not\vdash \lambda x.[x] : A^{\mathsf{H}} \to \Box_{\mathsf{L}} A \qquad \not\vdash \lambda x.x : A^{\mathsf{H}} \to A$$

In the latter, the base notion of usage at grade 1 (L) prevents us returning x.

A classic attack on data processing systems, that should also be guarded against, is via an *implicit flow*, in which some form of equality checking and control flow is used to leak information. This is known as a *control flow attack*. These are avoided in GrBase since the typing of **if-then-else** requires $r \leq 1$. Therefore the following is not valid:

$$\frac{x :_{\mathsf{L}} \mathbb{B} \vdash x : \mathbb{B} \quad \mathsf{H} \leq 1}{x :_{\mathsf{H}} \mathbb{B} \vdash \mathsf{if}\ x\ \mathsf{then}\ \mathtt{tt}\ \mathsf{else}\ \mathtt{ff} : \mathbb{B}}\ \textsc{If}\ (\textit{not valid})$$

Recall that $1 = \mathsf{L}$. Now, since $\mathsf{H} \not\leq \mathsf{L}$, this derivation is *not valid* as the second premise does not hold, and thus the program is rejected. This is the same solution as employed by Abel and Bernardy [2] and Choudhury et al. [18].

In Granule, the 2-point security semiring is given as the `Sec` type with constructor `Hi` for H and `Lo` for L. The equivalent of the above (non)example in Granule is thus also ill-typed, written as:

```
leak : Bool % Hi → Bool
leak x = if x then True else False   -- type error!
```

The `leak` function does not use the data-flow of variables to convey the input to the output, but rather uses the information gained via the conditional to reconstruct the input value. Since `Lo` is the 1 (top) element of the two-point security level lattice, we can however type the following:

```
public_not : Bool % Lo → Bool
public_not b = if b then False else True
```

$$\frac{}{(\lambda x.t_1)\ t_2 \rightsquigarrow [t_2/x]t_1}\ \textsc{SmallStepBeta} \qquad \frac{}{\mathsf{let}\ [x] = [t_1]\ \mathsf{in}\ t_2 \rightsquigarrow [t_1/x]t_2}\ \textsc{SmallStepboxBeta}$$

$$\frac{}{\mathsf{let}\ (x, y) = (t_1, t_2)\ \mathsf{in}\ t_3 \rightsquigarrow [t_1/x][t_2/y]t_3}\ \textsc{SmallStepprodBeta} \qquad \frac{}{\mathsf{let}\ () = ()\ \mathsf{in}\ t \rightsquigarrow t}\ \textsc{SmallStepunitBeta}$$

$$\frac{}{\mathsf{if}\ \mathtt{tt}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \rightsquigarrow t_1}\ \textsc{SmallStepIfTrue} \qquad \frac{}{\mathsf{if}\ \mathtt{ff}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \rightsquigarrow t_2}\ \textsc{SmallStepIfFalse}$$

(a) β-rules

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \textsc{SmallStepAppLeft} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathsf{let}\ [x] = t_1\ \mathsf{in}\ t_2 \rightsquigarrow \mathsf{let}\ [x] = t_1'\ \mathsf{in}\ t_2}\ \textsc{SmallStepcongLetL}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\mathsf{let}\ (x, y) = t_1\ \mathsf{in}\ t_2 \rightsquigarrow \mathsf{let}\ (x, y) = t_1'\ \mathsf{in}\ t_2}\ \textsc{SmallStepcongProd} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathsf{let}\ () = t_1\ \mathsf{in}\ t_2 \rightsquigarrow \mathsf{let}\ () = t_1'\ \mathsf{in}\ t_2}\ \textsc{SmallStepcongUnit}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \rightsquigarrow \mathsf{if}\ t'\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2}\ \textsc{SmallStepIfStep}$$

(b) Congruence rules

Fig. 2: Operational semantics for GrBase

We can also allow secret inputs to flow to secret outputs, e.g., as:

```
secret_not : Bool % Hi → Bool [Hi]
secret_not b = [if b then False else True]
```

Crucially here, the promotion on the outside of the conditional means that all the variable usage inside is scaled by `Hi`, which is the absorbing 0 element of the semiring, and thus the input here can be typed with `Hi` grade.

The original work on Granule introduced a different lattice for security levels, which did not prevent control flow attacks [46, Def 4.2]. This has since been removed from the language, replaced by the approach described in this paper (see, e.g., work by Hughes and Orchard on program synthesis from graded types which uses the above `Sec` semiring in Granule [31]).

3.2 Operational Semantics

Figure 2 gives the operational semantics for GrBase, which is standard and call-by-name, with the most interesting case being graded modal elimination.

A key property of the calculus is that of substitution admissibility, which is then used to establish syntactic type safety.

Lemma 1 (Admissibility of substitution). *If $\Gamma_1 \vdash t_1 : A$ and $\Gamma_2, x : r\ A \vdash t_2 : B$ then $\Gamma_1 + r \cdot \Gamma_2 \vdash [t_1/x]t_2 : B$.*

Proposition 1 (Syntactic type safety). *If $\Gamma_1 \vdash t_1 : A$ then either t_1 is a value or $t_1 \rightsquigarrow t_1'$ with $\Gamma_1 \vdash t_1' : A$.*

Other models have been proposed for capturing soundness of grading with regards an operational model, e.g., the heap semantics of Grad [18], also used by Marshall et al. [38], and the abstract machine of Abel and Bernardy [2]. Our focus here (and next) is however on a relational model to establish non-interference.

4 Semantic Model and Non-Interfering Coeffects

We define the semantics of types by building a logical relation model following Rajani et al. [55]. The key difference from the work of Rajani et al. is that our calculus is call-by-name rather than call-by-value, and performs a coeffect-based rather than an effect-based security analysis. Following their approach, the semantic model is split into two relations: a unary relation and a binary relation, the latter of which is key to relating terms based on grades and with relation to an observer (who has some security level).

The unary relation is fairly straightforward; it is only required to interpret secret (and thus potentially unrelated terms) in the two executions. The type-indexed family $[A]_{\mathcal{V}}$ defines the unary value interpretation as the set of values (irreducible terms) that inhabit the type A. Similarly, $[A]_{\mathcal{E}}$ defines the unary expression interpretation as the set of expressions (reducible terms) that inhabit the type A. Formally, $[A]_{\mathcal{V}}$ and $[A]_{\mathcal{E}}$ are defined in Figure 3 as mutually inductive definitions. For the value interpretation, we only explain the function and the graded types here, as the other cases are self explanatory. For the function type, $A^{s} \to B$, the values inhabiting its interpretation are lambda terms $\lambda x.t$ where the substitution of any term t' into the body is in the interpretation of the expression relation at the result type (due to the call-by-name semantics of our language). The interpretation of the graded modal type $\Box_r A$ is the set of terms which are the promotions of terms in the interpretation A. Note that, since the purpose of the grade r is to relate two values of type A, the unary relation completely ignores the grade. The importance of grades only shows up in the binary relation (as explained below). Finally, the $[\Gamma]$ relation defines an interpretation of the context Γ as a set of maps from variables to expressions (written as γ) that can be substituted for each variable.

The binary relation (Figure 4) formalises the key security requirements with respect to grades. Like the unary, the binary relation is defined for values and expressions. However, the key difference is that the binary interpretation of a type is a set of *pairs of terms*, as opposed to just a set of terms as in the unary

$$
\begin{aligned}
&\textit{(expression)} && [A]_{\mathcal{E}} = \{t \mid t \rightsquigarrow^{*} v \implies v \in [A]_{\mathcal{V}}\} \\
&\textit{(value)} && [\mathbb{B}]_{\mathcal{V}} = \{\mathtt{tt}, \mathtt{ff}\} \\
& && [\mathsf{Unit}]_{\mathcal{V}} = \{1\} \\
& && [A \otimes B]_{\mathcal{V}} = \{(t_1, t_2) \mid t_1 \in [A]_{\mathcal{V}} \wedge t_2 \in [B]_{\mathcal{V}}\} \\
& && [A^{s} \to B]_{\mathcal{V}} = \{\lambda x.t \mid (\forall t'.\ [t'] \in [\Box_s A]_{\mathcal{E}} \implies [t'/x]t \in [B]_{\mathcal{E}})\} \\
& && [\Box_r A]_{\mathcal{V}} = \{[t] \mid t \in [A]_{\mathcal{E}}\} \\
&\textit{(context)} && [\Gamma] = \{\gamma \mid x :_r A \in \Gamma \wedge [\gamma(x)] \in [\Box_r A]_{\mathcal{E}}\}
\end{aligned}
$$

Fig. 3: Unary value, expression and context relations

$$
\begin{aligned}
\textit{(expression)}\quad [\![A]\!]^{adv}_{\mathcal{E}} &= \{(t_1,t_2) \mid t_1 \rightsquigarrow^* v_1 \ \wedge \ t_2 \rightsquigarrow^* v_2 \implies (v_1,v_2) \in [\![A]\!]^{adv}_{\mathcal{V}}\} \\
\textit{(value)}\quad [\![\mathbb{B}]\!]^{adv}_{\mathcal{V}} &= \{(\mathtt{tt},\mathtt{tt}),(\mathtt{ff},\mathtt{ff})\} \\
[\![\mathsf{Unit}]\!]^{adv}_{\mathcal{V}} &= \{(1,1)\} \\
[\![A \otimes B]\!]^{adv}_{\mathcal{V}} &= \{((t_1,t_2),(t_1',t_2')) \mid (t_1,t_1') \in [\![A]\!]^{adv}_{\mathcal{V}} \wedge (t_2,t_2') \in [\![B]\!]^{adv}_{\mathcal{V}}\} \\
[\![A^s \to B]\!]^{adv}_{\mathcal{V}} &= \{(\lambda x.t_1, \lambda y.t_2) \mid (\forall t,t'.([t],[t']) \in [\![\Box_s A]\!]^{adv}_{\mathcal{E}} \implies \\
&\qquad ([t/x]t_1,[t'/y]t_2) \in [\![B]\!]^{adv}_{\mathcal{E}}) \\
&\qquad \wedge (\forall t.[t] \in [\Box_s A]\mathcal{E} \implies [t/x]t_1 \in [B]\mathcal{E}) \\
&\qquad \wedge (\forall t.[t] \in [\Box_s A]\mathcal{E} \implies [t/y]t_2 \in [B]\mathcal{E}) \ \} \\
[\![\Box_r A]\!]^{adv}_{\mathcal{V}} &= \begin{cases} \{([t_1],[t_2]) \mid (t_1,t_2) \in [\![A]\!]^{adv}_{\mathcal{E}}\} & \textit{when} \ \ r \leq adv \\ \{([t_1],[t_2]) \mid t_1 \in [A]\mathcal{E} \wedge t_2 \in [A]\mathcal{E}\} & \textit{when} \ \ \neg(r \leq adv) \end{cases} \\
\textit{(context)}\quad [\![\Gamma]\!]^{adv} &= \{(\gamma_1,\gamma_2) \mid x :_r A \in \Gamma \ \wedge \ [\gamma_1(x),\gamma_2(x)] \in [\![\Box_r A]\!]^{adv}_{\mathcal{E}}\}
\end{aligned}
$$

Fig. 4: Binary expression, value and context relations

case. The binary interpretation is also parameterised by a grade representing the adversary level *adv* against whom we desire *indistinguishability* of secrets.

We explain the binary expression relation first: $[\![A]\!]^{adv}_{\mathcal{E}}$ relates two expressions of type A when the values obtained by reducing them can be related under the binary value interpretation at the same type. Since we only relate the resulting values when both the expressions terminate, our logical relation internalises what is known as 'termination-insensitive non-interference' [64]. We believe that our current development can be generalised to a termination sensitive version of non-interference by additionally proving co-termination of the two executions. We leave that part as an interesting direction for future work.

For the binary value interpretation ($[\![A]\!]^{adv}_{\mathcal{V}}$), like in the unary case, we focus only on the function and the graded types here. The interpretation of the function type $A^s \to B$ relates two λ-expressions, when given related expressions at an input type ($\Box_s A$) the bodies of the λ-expression with the substitutions should be related at the output type B. The additional unary clauses are required for technical purposes (as we desire an invariant that any two terms that are related by the binary relation must also be separately in the unary relation at the same type). The value interpretation for the graded type $\Box_r A$ is now the most critical point in the model for security. It relates two promoted terms $[t_1]$ and $[t_2]$ as follows: if the grade r is less than or equal to the adversary level *adv* (i.e., $r \leq adv$) then t_1 and t_2 must be related by the binary expression relation at A, otherwise t_1 and separately t_2 are just in the unary expression interpretation of A and thus cannot be distinguished; an adversary who is at the low security level (L) cannot distinguish values which are designated as secret (at the high security level) by being in inside a H-graded box, since $\neg(\mathsf{H} \leq \mathsf{L})$.

Finally, $[\![\Gamma]\!]^{adv}$ defines the binary relation for relating substitutions in the two runs, reusing the interpretation of the graded modality to capture grading.

4.1 Fundamental Theorems and Non-Interference

A key property of our model is the so-called 'fundamental theorem' which connects the syntactic type system with its semantic interpretation: any expression subject to a syntactic typing is member of the semantic typing. We define a unary (Theorem 1) and a binary (Theorem 2) version of the fundamental theorem, corresponding to the two logical relations.

Theorem 1 (Unary fundamental theorem). *For all well-typed terms $\Gamma \vdash t : A$, if $\gamma \in [\Gamma]$ then $\gamma\ t \in [A]_{\mathcal{E}}$ (where the $\gamma\ t$ applies a substitution of the mapping from variables to terms given by γ).*

Proof. (`utheorem`, `UnaryFundamentalTheorem.agda`) By induction on typing.

The unary fundamental theorem says that if t is a term of type A in a context Γ, and γ is a unary substitution in the interpretation of Γ then $\gamma\ t$ is in the unary expression relation at the type A.

The binary fundamental theorem instead leverages the binary relation, which is where we get the interaction between an external observer (the 'adversary') and the grading. It is at this stage that we need additional structure to enable the binary fundamental lemma as the crucial step to non-interference. This additional structure delineates a subclass of semirings which we call *non-interfering*:

Definition 4 (Non-interfering semiring). *A non-interfering semiring $\mathcal{R}$ is a pre-ordered semiring with four additional axioms:*

- *lax idempotence*[10] *of multiplication: $r \cdot r \leq r$*
- *antisymmetry: $(r \leq s) \wedge (s \leq r) \implies r = s$*
- *multiplicative unit is minimal (bottom): $1 \leq r$*
- *additive unit is maximal (top): $r \leq 0$*

Proposition 2 (Derived properties). *For a non-interfering semiring, then we have the following derived properties:*

- *(decreasing $+$) If $r_1 \leq r_2$ then $(r_1 + s) \leq r_2$*
  (which follows from 0 being the top and the usual semiring properties of $+$ monotonicity and 0 being the additive unit).
- *(increasing $\cdot$) If $r_1 \leq r_2$ then $r_1 \leq (s \cdot r_2)$ and $r_1 \leq (r_2 \cdot s)$*
  (which follows from 1 being the bottom and the usual semiring properties of $\cdot$ monotonicity and 1 being the multiplicative unit).

From these two properties we can conversely derive the minimality of 1 and the maximality of 0 respectively.

[10] There seems to be no universally accepted name for this property. *Weak* idempotence is used in [4] to denote the property $r^4 = r^2$, hence we opted for the modifier *lax*.

Proposition 3 (2-point security lattice is non-interfering). *The semiring induced by the lattice on $\mathcal{L} = \{H, L\}$ with $L \leq H$, where $0 = H$ and $1 = L$, is non-interfering, with lax idempotence of multiplication trivial since $\cdot = \vee$, with the ordering providing antisymmetry, minimality of 1, and maximality of 0.*

We can now give the binary fundamental theorem on this subclass of semirings:

Theorem 2 (Binary fundamental theorem). *For all well-typed terms $\Gamma \vdash t : A$ with a semiring $\mathcal{R}$ parameterising the calculus that is* non interfering*, if $(\gamma_1, \gamma_2) \in [\![\Gamma]\!]^{adv}$ then $(\gamma_1\ t, \gamma_2\ t) \in [\![A]\!]^{adv}_{\mathcal{E}}$ (where the $\gamma_1\ t$ and $\gamma_2\ t$ applies a multi-substitution of the mapping from variables to terms given by γ_1 and γ_2).*

The binary fundamental theorem is similar to the unary, but relates the syntactic typing to the semantic typing according to the binary expression relation. Both of these theorems are proved by induction on the typing derivation.

Proof. Formalised (`biFundamentalTheorem`, `BinaryFundamentalTheorem.agda`).

The theorem follows by induction over typing derivations. We sketch some of the pertinent cases here which show where the axioms of the *non-interfering semiring* are used. (Note uses of axioms are typeset in green.) The interesting cases typically occur when incoming context has a grade that is not observable by the adversary (i.e., some r such that $\neg(r \leq adv)$) but some premise requires a context with a grade s that is observable by the adversary (i.e., some s such that $s \leq adv$).

- (var) with typing $0 \cdot \Gamma, x : 1\ A \vdash x : A$, and goal $[\![A]\!]^{adv}_{\mathcal{E}}$.
  Thus the interpretation of the context in the assumption of the theorem provides $([t_1], [t_2]) \in [\![\Box_1 A]\!]^{adv}_{\mathcal{E}}$ with two possible cases for its meaning:
  - $1 \leq adv$ and $(t_1, t_2) \in [\![A]\!]^{adv}_{\mathcal{E}}$ satisfying the goal of the theorem.
  - $\neg(1 \leq adv)$ and thus $t_1 \in [A]_{\mathcal{E}} \wedge t_2 \in [A]_{\mathcal{E}}$ which cannot satisfy the goal of $[\![A]\!]^{adv}_{\mathcal{E}}$. However there is a contradiction here since 1 is the bottom element ($\forall r.1 \leq r$) in a non-interfering semiring, and thus this case holds by *ex falso quod libet*.
- (app) with typing $\Gamma_1 + r \cdot \Gamma_2 \vdash t_1\ t_2 : B$ and premises $\Gamma_1 \vdash t_1 : A^r \to B$ and $\Gamma_2 \vdash t_2 : A$.
  Thus we have a context interpretation $(\gamma_1, \gamma_2) \in [\![r \cdot \Gamma_1 + \Gamma_2]\!]^{adv}$ from which we need to derive a context $(\gamma_1', \gamma_2') \in [\![\Gamma_1]\!]^{adv}$ to compute the left-hand side t_1. For every x such that $x :_{r_1} A \in \Gamma_1$ and $x :_{r_2} A \in \Gamma_2$ then this requires us to map from $([t_3], [t_4]) \in [\![\Box_{r_1 + r \cdot r_2} A]\!]^{adv}_{\mathcal{E}}$ to $([t_3], [t_4]) \in [\![\Box_{r_1} A]\!]^{adv}_{\mathcal{E}}$ in the interpretation of the context in order to apply induction over the first typing premise. In the case where $\neg(r_1 + r \cdot r_2 \leq adv)$ but $r_1 \leq adv$ then we have a contradiction since $r_1 \leq adv$ implies $r_1 + r \cdot r_2 \leq adv$ by the decreasing property of $+$ (i.e., that 0 is the top element, see Proposition 2). A similar use of this property occurs in every other case where the context need splitting, i.e., expressions comprising more than two subterms.

– (pr) with typing $r \cdot \Gamma \vdash [t] : \Box_r A$ and premise $\Gamma \vdash t : A$.
For all $x :_s B \in \Gamma$ then context interpretation contains some $([t_1], [t_2]) \in [\![\Box_{(r \cdot s)} B]\!]^{adv}_{\mathcal{E}}$ from which we need that $([t_1], [t_2]) \in [\![\Box_s B]\!]^{adv}_{\mathcal{E}}$ in order to induct on the premise.
In the case where $r \leq adv$ but $\neg(r \cdot s \leq adv)$ and thus $[\![\Box_{(r \cdot s)} B]\!]^{adv}_{\mathcal{E}} = [B]_{\mathcal{E}} \wedge t_2 \in [B]_{\mathcal{E}}$ then we have $\neg(s \leq adv)$ by lax idempotence of $*$ (and a proof by contradiction) and thus $[\![\Box_s B]\!]^{adv}_{\mathcal{E}} = [B]_{\mathcal{E}} \wedge t_2 \in [B]_{\mathcal{E}}$ which is satisfied by the context here.
– (if) with typing $(r \cdot \Gamma_1) + \Gamma_2 \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : A$ and $r \leq 1$.
For all $x :_{r_1} B \in \Gamma_1$ and $x :_{r_2} B \in \Gamma_2$ then context interpretation contains some $([t_1], [t_2]) \in [\![\Box_{(r \cdot r_1 + r_2)} B]\!]^{adv}_{\mathcal{E}}$.
In the case where $\neg((r \cdot r_1) + r_2) \leq adv$ then due to the typing requiring $r \leq 1$ it follows that $\neg r_1 \leq adv$ by, i.e.,

(0)	$\neg((r \cdot r_1) + r_2) \leq adv$	(*assumption*)
(1)	$r_1 \leq adv$	(*assumption*)
(2)	$r \leq 1$	(*typing*)
(3)	$r \cdot r_1 \leq adv \cdot 1$	(*monotone* $\cdot$ (1), (2))
(4)	$r \cdot r_1 \leq adv$	(*multiplicative unit* (3))
(5)	$((r \cdot r_1) + r_2) \leq adv$	(*decreasing* $+$ (4))
(6)	$\bot$	($\neg$-*elim* (0),(5))
(7)	$\neg(r_1 \leq adv)$	($\neg$-*intro* (1),(6))

Thus the incoming context interpretation suffices for the context interpretation of t_1.

Note that the proof does not make use of the antisymmetry property yet, which instead arises in the final central theorem:

Theorem 3 (Non-Interference). *For all judgments* $x :_s A \vdash t : \Box_{adv} \mathbb{B}$ *where* $adv \leq s$ *and* $adv \neq s$ *then given* $\emptyset \vdash v_1 : A$ *and* $\emptyset \vdash v_2 : A$ *then* $[v_1/x]t \equiv [v_2/x]t$.

(where $\equiv$ is full β-equality of terms, derived from the operational semantics).

Proof. (`nonInterference`, `NonInterFerence.agda`) We sketch some interesting details here, but the binary fundamental theorem does most of the heavy lifting.

First, promote the typing of v_1 and v_2, i.e., to $\emptyset \vdash [v_1] : \Box_s A$ and $\emptyset \vdash [v_2] : \Box_s A$ and to both apply the binary fundamental theorem, yielding:

$$(\emptyset, \emptyset) \in [\![\emptyset]\!]^{adv} \implies ([v_1], [v_1]) \in [\![\Box_s A]\!]^{adv}_{\mathcal{E}} \tag{2}$$
$$(\emptyset, \emptyset) \in [\![\emptyset]\!]^{adv} \implies ([v_2], [v_2]) \in [\![\Box_s A]\!]^{adv}_{\mathcal{E}} \tag{3}$$

The antecedents are trivially satisfied and both (2), (3) are each equivalent to:

$$([v_1], [v_1]) \in ([\Box_s A]_{\mathcal{E}} \times [\Box_s A]_{\mathcal{E}})$$
$$([v_2], [v_2]) \in ([\Box_s A]_{\mathcal{E}} \times [\Box_s A]_{\mathcal{E}})$$

by the meaning of $[\![\Box_s A]\!]^{adv}_{\mathcal{E}}$ and since $adv \leq s$ from the premise and the fact that if we had also $s \leq adv$ then by antisymmetry it would follow $adv = s$ which is

contradicted by the premise that $adv \neq s$. We then apply the binary fundamental theorem with $(\{x \mapsto v_1\}, \{x \mapsto v_2\}) \in [\![x :_s A]\!]^{adv} \in [x :_s A] \times [x :_s A]$ to obtain $([v_1/x]t, [v_2/x]t) \in [\![\Box_{adv}\mathbb{B}]\!]^{adv}_{\mathcal{E}}$. By reflexivity of $\leq$, such that $adv \leq adv$, this interpretation is necessarily equal to $[\![\mathbb{B}]\!]^{adv}_{\mathcal{E}}$ and thus by the definition of the binary fundamental lemma $[v_1/x]t$ and $[v_2/x]t$ reduce to terms v and v' such that $(v, v') \in [\![\mathbb{B}]\!]^{adv}_{\mathcal{V}}$ which necessarily means $v \equiv v'$. □

Remark 2. The definition of non-interfering semirings (Defn. 4) permits the trivial, singleton semiring where $0 = 1$. This semiring would appear to permit interference since it conflates 0 and 1. However, the non-interference theorem assumes $adv \neq s$, thus the theorem holds trivially for the singleton semiring.

Another useful non-interfering semiring is the lattice of sets over labels:

Example 4 (Set semiring). Let $\mathcal{T}$ be a set of arbitrary abstract labels from which we form a semiring **Set** of sets generated by $\mathcal{P}(\mathcal{T})$ (the powerset of $\mathcal{T}$) under subset inclusion with multiplicative structure $\cdot := \cup$ with $1 := \emptyset$ and additive structure $+ := \cap$ with $0 := \top$.

This gives one of the staples of secure information flow tracking [19], which serves as the (hard-coded) basis of other information flow control systems such as Flow Caml [58]. This semiring is also supported in Granule, where labels are defined via a sum type with unit constructors. As a concrete example, we define a datatype `Privilege` to model privileged data sources on a mobile device:

```
data Privilege = Camera | Contacts | Location | Microphone

myLocation : String [{Location}]
myLocation = ["051.28N 001.08E"]

myFavouriteContact : String [{Contacts}]
myFavouriteContact = ["Alice"]
```

We have a value `myLocation` that is marked as needing location access via its graded modal type and a value `myFavouriteContact` that is marked as needing access to contacts. Use of these values in other parts of the program propagates their requirements, as shown by these two examples:

```
honest : String [{Contacts, Location}]
honest =
  let [x] = myLocation;
      [y] = myFavouriteContact
  in [stringAppend x y]

dishonest : String [{Contacts}]
dishonest =
  let [x] = myLocation;
      [y] = myFavouriteContact
  in [stringAppend x y]    -- type error!
```

Here, `honest` requires the caller to provide a privilege context including at least contacts and location access, i.e. the union of the privileges of `myLocation` and

`myFavouriteContact`. A type error is raised for `dishonest`, as it fails to require location access.

Dually to the **Set** semiring defined above, we have another valid (non-interfering) semiring given by: $\mathbf{Set}^{\mathsf{op}} := (\mathcal{R} := \mathcal{P}(\mathcal{T}), \cdot := \cap, 1 := \top, + := \cup, 0 := \emptyset, \leq := \supseteq)$. We leave its interpretation open.

4.2 Comparison with related Results in the Literature

Abel and Bernardy (2021). From Abel and Bernardy's 'unified view' [2], §4.3 considers 'informational applications' of graded types in which "it does not matter how many times variables are used, but rather *in which context* they are used." In this setting they restrict themselves to semirings in which the $+$ is the meet derived from an ordering and multiplication is the join, explicitly mentioning that this means that multiplication is idempotent (which exceeds the lax idempotence required by our definition). Their system is defined over a ringoid– a semiring with a meet-semilattice– with a partial order defined by the meet, hence providing the antisymmetry property. In §4.3.2. they assume that everything "above 1 in the lattice is secret", and give the 2-point H/L example, noting "the construction generalises however to any lattice of information modalities as specified". Any semiring in their class of 'informational semirings' is non-interfering[11] since defining the $+$ as the $\wedge$ and $\cdot$ as $\vee$ implies the $+$ is decreasing and $\cdot$ is increasing, and thus by Proposition 2 we have that 0 is maximal and 1 is minimal.

Based on a relational model, they prove a non-interference theorem (Theorem 7.10) for terms of the type (in our syntax) $f : A^1 \to B^0 \to \mathbb{B}$ showing that $[\![f]\!]\ u\ b_1 = [\![f]\!]\ u\ b_2$ for $u \in [\![A]\!]$ and $b_1, b_2 \in [\![B]\!]$. This theorem resembles ours but is more specialised, focusing just on the 0 and 1 grades (where their result type $\mathbb{B}$ can be read equivalently as $\Box_1\mathbb{B}$).

It should be noted that their work provides a more general relational model which can be used to consider properties beyond just non-interference. On the other hand, the work presented here has a more general non-interference result and defines a larger class of semirings with this property.

Abel et al. (2023). Formalising a graded modal dependent type theory, Abel et al. (2023) provide a graded system with many features, generalising the core calculus described here [3]. They define a semiring-based grading algebra over a partial ordering, and requiring the existence also of a meet operation. They consider an in-depth case study looking at *erasure*, in which grading is used to track which parts of a program are not used and can be erased—a similar property to non-interference: erasable arguments are non-interfering. They define a subclass of grading structures, said to have a *well-behaved zero* (similar to what Moon et al. [43] call a 'quantitative semiring') if:

– Additive is positive: if $p + q = 0$ then $p = 0$ and $q = 0$;

[11] (`informationalImpliesNonInterfering, Semiring.agda`)

- Meet is positive: if $p \wedge q = 0$ then $p = 0$ and $q = 0$;
- The zero-product property holds: if $p \cdot q = 0$ then $p = 0$ or $q = 0$;
- Non-triviality $0 \neq 1$.

They point out that the second property implies that 0 is maximal. They establish a logical relations model with a fundamental theorem for erasure for those well-behaved zero algebras (Theorem 6.10), with a thorough formalisation to go with it. From this they establish non-interference for the 2-point semiring and point out that more fine-grained lattices with 1 as minimal can also give the same result. Again, this result overlaps with that here, but requiring more structure compared to our 'leaner', more general approach.

5 Practical Example of Mandatory Access Control

In this section we explore a practical case study of our security coeffects in Granule. These examples take inspiration from Haskell's Mandatory Access Control (MAC) library [57], though the code in that paper is written in a monadic effect-based style. We instead represent these local side-effects using Granule's linearly-typed session channels [39] (which are based on the GV calculus [22]). This provides an interface for channels `LChan : Protocol → Type` where:

```
data Protocol = Send Type Protocol | Recv Type Protocol | End
```

with the following (non-monadic) linear functions:

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

The `Dual` type family is defined internally as follows:

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

The first example below defines a function `attemptLogin` which will authenticate a (public) user ID and (secret) password, logging each attempt by sending only the ID over a public channel to be printed. Note that the boolean result is given in a `Hi` context, meaning that the result can only be used at this level of confidentiality.

```
attemptLogin : Int
            → String % Hi
            → LChan (Send Int End) -- logging channel
            → Bool [Hi]
attemptLogin uid pass c =
  let c = send c uid;
      () = close c
  in validateLogin uid pass
```

Here, the auxiliary `validateLogin` function has the type:

```
validateLogin : Int → String % Hi → Bool [Hi]
```

Thus, we take in the ID and password and return the result stating whether the login has been validated or not. As one might expect, this result must be secret, since the password was itself secret data.

In the `attemptLogin` function, if we tried to leak secret data by logging `pass` over the channel rather than `uid`, this would be disallowed by the type system.

We can now extend this example and write a function `doQuery`, which authenticates a user and then performs a query on secret data, returning, in a secret context, a string representing the query result if the user's password is correct and otherwise returning an error.

```
doQuery : Int                                        -- uid
        → String % Hi                               -- password
        → (LChan (Recv (String [Hi]) End))          -- secret data!
        → (Either Error String) [Hi]
doQuery uid pass secret =
  let c = forkLinear logLoginAttempt in
  let [login] : Bool [Hi] = attemptLogin uid pass c in
  [ if login
    then
      Left AuthErr
    else
      let [secr] : String [Hi] = getSecret secret
      in Right secr
  ]
```

The auxiliary `logLoginAttempt` function here is of type

```
logLoginAttempt : LChan (Recv (Int [Lo]) End) → ()
```

Since this function is public-facing, it would not be possible to log secret data such as the password or even the boolean result stating whether the login was successful; all we can log is that an attempt to login has occurred.

Note also that once we discriminate on the result of `attemptLogin`, we enter a secret context, and so even though `uid` is still public data it would no longer be possible to log this or anything else after we have reached this point. Thus, an implicit leak is prevented. The case expression has the same typing as conditional shown in Section 3.

The next example is based on the main case study demonstrated in Haskell in the MAC paper. Here, we define a function `isCommonPW` which checks whether a given password is in a standard list of common passwords. The list is public information, but the password given to the function is of course secret data, and so the result from the function must also be secret.

```
isCommonPW : String % Hi → List String → Bool [Hi]
isCommonPW password commonPasswords =
    [ case commonPasswords of
        Nil → False;
        Cons x xs →
          case strEq x password of
```

```
            [True] →
              let () = drop @(List String) xs in True;
            [False] →
              let [isCommon] = isCommonPW password xs in isCommon
      ]
```

where `strEq : String % Hi → String % Hi → Bool [Hi]` is omitted for brevity. The list of common passwords (second parameter) is public, against which we compare our secret password recursively, returning a secret result about whether the password is in the common list or not. This makes use of a few additional Granule features, like the `drop @(List String)` feature which, based on the type after `@`, derives a discarding function for the list of strings [32]. The main take-away here is that security reasoning with coeffects scales to practical examples involving algebraic data types and recursion.

6 Discussion and Conclusion

Section 2 considered the background related work on coeffects and graded type systems in depth. Section 4 also considered closely the most pertinent related work here with regards the coeffect-style to IFC. To round out the related work discussion, we consider relationship with the effect-style approach to IFC.

6.1 Alternate Presentations of Information-Flow Control

Fine-grained information-flow type systems like SLAM [29], Flow Caml [54] also use an (annotated) modal type constructor to associate labels with every type. For example, a pair of type A and B has labels associated to each component and the pair as a whole, e.g., $(A^{L1} \times B^{L2})^{L3}$. GrBase on the other hand has a coeffectful approach to labeling, explicit labels on inputs are mandatory, but explicit labels on outputs are not: instead the output is essentially equivalent to being graded at 1.

In contrast, coarse-grained type systems like DCC [1] associate and track dependencies for a language without side-effects, with annotations on *monadic constructors* (essentially extending Moggi's monadic meta language [42]). Type systems like HLIO [16] and CG [55, 56] extend the coarse-grained approach for languages with higher-order mutable state. However, both HLIO and CG use a different modal type constructor to associate labels with a type, and a monad for tracking the flows. Unlike all these systems, GrBase uses (comonadic) coeffect grades for explicit labeling. As a result, our logical relation model for GrBase draws inspiration from the model of the fine-grained system FG due to Rajani et al. [55].

A modern perspective on DCC is that it leverages a *graded monad* to package the monad along with its structured annotations (to see this, one might follow the thread from Wadler and Thiemann's 'marriage of effects and monads' forming a lattice of monads [67], to the more general notion of graded monads; Mycroft et al. trace the history of these developments and the overall paradigm [44]).

Algehed examines the monadic structure of DCC in more detail [5], showing that DCC embeds a graded monad T_ℓ with grades ℓ drawn from the two-point lattice $\mathsf{L} \leq \mathsf{H}$ (also used here) but with additional structure. This additional structure amounts to a kind of distributive law that provides $T_\ell T_{\ell'} A \to T_{\ell'} T_\ell A$. Choudhury conducts a similar but more thorough study of the structure of DCC from both a graded monadic vs. graded comonadic perspective, noting that "the monadic aspect of dependency analysis might be just half of the story [...] experience shows us that security constraints can be enforced not only by restricting outflow but also by restricting inflow" [17]. Choudhury shows, similarly to Algehed, that DCC is not just graded monadic but encodes a distributive law in its calculus. He goes on to show that this distributive law can be captured by a structure which is *both* a graded monad and graded comonad, whose operations are mutually inverse, with a derivation $T_\ell T_{\ell'} A \xrightarrow{\mu} T_{\ell \vee \ell'} A \equiv T_{\ell' \vee \ell} A \xrightarrow{\delta} T_{\ell'} T_\ell A$. In essence, DCC has both graded monadic and graded comonadic aspects. Here we focussed on just the graded comonadic side of this equation via coeffects and the coeffect-graded modality. Granule however allows the missing graded monadic part to be recovered by its 'double unboxing' pattern matching [46], with:

```
joinSec : ∀ {a : Type, l l' : Sec} . (a [l]) [l'] → a [l * l']
joinSec [[x]] = [x]
```

Following Choudhury's analysis, adding this as a primitive to our GrBase calculus here would yield a system which is equivalent in power to DCC [17]. Choudhury's calculus has a slightly different form to our own (with no linearity nor graded assumptions or functions, but explicit graded comonad operations instead), so a little more work is needed to formalise this result, which we leave as further work. Our system is yet more general though by the use of the general class of non-interfering semirings.

6.2 Declassification

The approach to IFC here features non-interference, but is extremely restrictive: there cannot be any flow from a high security level to a lower. In reality, we often want software systems to leak a small amount of information in a controlled way. This is known as *declassification* [65, 45].

Our existing notion of a non-interfering semiring permits a generalisation of the 2-point lattice considered here which, along with some built-in primitives, can be used to capture the number of bits declassified in a computation as a further instance of non-interfering semirings:

Example 5 (Declassification semiring). The $\mathsf{L} \leq \mathsf{H}$ semiring can be generalised to a semiring over $\mathcal{R} = \mathbb{N} \cup \{\omega\}$ to capture the *number of declassified bits*, where L (which was the multiplicative unit 1) is replaced by ω to represent an arbitrarily large number of bits being declassified and H (which is 0) being replaced by $0 \in \mathbb{N}$ to represent 0-bits being declassified. In between are all possible representations of a finite number of bits being declassified with the pre-ordering defined by $\omega \leq \ldots \leq 1 \leq 0$. Addition is then the maximum on $\mathbb{N} \cup \{\omega\}$, i.e., the number of

bits declassified by two subterms is the largest number (which according to this ordering is the 'meet' $\wedge$, since the ordering is going down towards infinity) and multiplication is then the minimum ('join' $\vee$ on this ordering), i.e., the number of bits declassified by a composed term is the smaller number.

A declassification primitive then acts as the operation of this graded modality, declassifying a precise number of bits for various data types, for example, for integers, there is a primitive function of indexed type:

```
declassifyInt : ∀ {n : Nat} . Nat n → Int [n] → Int
```

Then, a program declassifying 3-bits and then 5-bits of information from its argument overall has to be typed as declassifying 5-bits: the maximum number of declassified bits (via $+$). A program declassifying 3-bits of its argument and then composing this with a function which declassifies 5-bits of its argument declassifies only 3-bits of information overall, i.e., the minimum (via $\cdot$).

This declassification semiring is a non-interfering semiring.

Future work is to consider a quantitative non-interference theorem that allows reasoning about such declassification operations and their effect on non-interference; we need some model that can account for the above declassification operation and then a generalisation to our non-inference theorem here. Specifically we would like to explore how to integrate work on declassification from the literature, such as Relative Secrecy [65]. This is further work.

An alternate approach to declassification by Hainry and Péchoux requires that declassification operations can only happen outside of loops [27], preventing control-flow attacks from iteratively transferring information from secret to public data. We can emulate this idea in Granule, not by restricting the language or using some additional static analysis, but by leveraging grading to enforce a linearity constraint.

We sketch out a more tentative implementation of this idea by introducing an abstract data type `DeclassCap` capturing a declassification capability. This could then be consumed by the following primitive function, which compares two secret strings, producing a result which is public, but requiring a declassification capability to be consumed once:

```
equality : DeclassCap % (1 : Nat)
         → String % Hi → String % Hi → Bool [Lo]
```

Remark 3. Any definition that allows for some declassification needs to be provided as an axiom, i.e. a primitive in the compiler, as it is not internally derivable. And so much the better, since this technically introduces a violation of non-interference!

There would be no constructor for `DeclassCap`, instead a special compilation mode would expect the entry-point of a program to be a function receiving a graded declassification capability that can be used a finite number of times:

```
mainDeclass : ∀ {n : Nat} . DeclassCap % n → Int [Hi]
```

In this way we use grading to restrict declassification to a specified number of comparisons only.

6.3 Critical Analysis and Flat Coeffects for Security?

One criticism of the approach described here, for example considering the two-point lattice, is that conflating high security (H) with non-usage (0 of the semiring) leads to unintuitive proofs and typing. For instance, the derivation below (equation 4) is valid in GrBase because if we can prove B from 0 copies of A then we can also prove $\Box_{\mathsf{L}} B$ from 0 copies of A:

$$\frac{x : [A]_{\mathsf{H}} \vdash t : B}{x : [A]_{\mathsf{H}} \vdash t : \Box_{\mathsf{L}} B} \tag{4}$$

since the context here has the grade calculated by $\mathsf{L} \cdot \mathsf{H} = \mathsf{L} \vee \mathsf{H} = \mathsf{H}$.

This concluding type on its own looks very worrying for non-interference: we have a computation that takes a high-security (secret) input and produces an output which can be used in a low-security (public) context. However, the fact that $0 = \mathsf{H}$, coupled with the rest of the structure of the type system, means that non-interference holds (as proved in Section 4). Instead, we could imagine a system in which we avoid conflating 0 and H but require that a term producing a low-security result requires all its dependencies to be low security.

Thus, whilst the type system is sound (it gives us non-interference), the grades require some additional thought. Furthermore, the system severely restricts expressivity in order to gain non-interference and avoid control-flow attacks: as seen, we can only guard control-flow on a secret value if the entire control-flow expression is promoted, e.g.

```
pnot : Bool % Hi → Bool [Hi]
pnot x = [if x then True else False]
```

Instead, we ask, what would it take it allow the following alternate implementation, which is not well-typed under the system of this paper?

```
pnot' : Bool % Hi → Bool [Hi]
pnot' x = if x then [True] else [False]
```

One approach could be to return to the *flat coeffects* of the original ICALP 2013 paper by Petricek, Orchard, and Mycroft [52] adding an ambient coeffect affected by control-flow. Thus guarding a conditional with a secret value would introduce an ambient security requirement into the context which must be respected by each branch. Exploring this is future work.

6.4 Concluding Remarks

The coeffect types paradigm set out by Alan Mycroft and his students sought to capture various context-dependent analyses via a unified type system. This sprung from trying to dualise effect types, but later work by Alan and his students circled back to effects to reconsider them in the light of the lessons learned

from the coeffect system and its semantics [44]. This lead to the umbrella approach of *graded modal types* or more generally just *graded types*. As we've seen in the GrBase style: we do not necessarily need the explicit graded modality, but can instead have a type system with pervasive grading. Our focus here was on security, an application picked up in a number of other graded types works [2, 3, 21], but many questions remain about how to generalise these ideas to declassification for more practical programming. It would also be interesting to explore how to capture the notion of *integrity* (dual in a sense to security/confidentiality) in a more general manner than has been previously discussed [37].

An opportunity presents itself for the further impact of these ideas through the GHC Haskell compiler, which already implements a simple graded coeffect type system internally to enable its `LinearTypes` extension [8]. Perhaps some day in the future there may even be a `SecurityTypes` or `CoeffectTypes` extension that generalises that implementation, building on the more general systems laid down by Mycroft et al. and as seen in the Granule language.

Acknowledgments This work was partly supported by EPSRC grant EP/T013516/1 (*Verifying Resource-like Data Use in Programs via Types*). Orchard also received support through Schmidt Sciences, LLC. Rajani was supported in part by the EPSRC grant EP/Y003535/1 (*Type-based information declassification and its secure compilation*). Rajani and Orchard also received support through ARIA's Safeguarded AI programme. Vollmer was supported in part by EPSRC grant EP/X021173/1.

The authors warmly thank Professor Alan Mycroft for all his inspiration, wisdom, patience, enthusiasm, and ideas that have gone into this work and the many other works described here. Enjoy your retirement Alan, you've earned it!

Bibliography

[1] Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: Appel, A.W., Aiken, A. (eds.) POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999. pp. 147–160. ACM (1999). https://doi.org/10.1145/292540.292555

[2] Abel, A., Bernardy, J.: A unified view of modalities in type systems. Proc. ACM Program. Lang. **4**(ICFP), 90:1–90:28 (2020). https://doi.org/10.1145/3408972

[3] Abel, A., Danielsson, N.A., Eriksson, O.: A graded modal dependent type theory with a universe and erasure, formalized. Proc. ACM Program. Lang. **7**(ICFP), 920–954 (2023). https://doi.org/10.1145/3607862

[4] Abera, T., Yitayew, Y., Wasihun, D., Kolluru, V.: Ordered weak idempotent rings. Palestine Journal of Mathematics **13**(3) (2024)

[5] Algehed, M.: A Perspective on the Dependency Core Calculus. In: Alvim, M.S., Delaune, S. (eds.) Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 24–28. ACM (2018). https://doi.org/10.1145/3264820.3264823

[6] Atkey, R.: Syntax and semantics of quantitative type theory. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 56–65. ACM (2018). https://doi.org/10.1145/3209108.3209189

[7] Benton, N., Bierman, G., De Paiva, V., Hyland, M.: Linear λ-calculus and categorical models revisited. In: Computer Science Logic: 6th Workshop, CSL'92 San Miniato, Italy, September 28–October 2, 1992 Selected Papers 6. pp. 61–84. Springer (1993)

[8] Bernardy, J., Boespflug, M., Newton, R.R., Jones, S.P., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. Proc. ACM Program. Lang. **2**(POPL), 5:1–5:29 (2018). https://doi.org/10.1145/3158093

[9] Bianchini, R., Dagnino, F., Giannini, P., Zucca, E.: A Java-like calculus with heterogeneous coeffects. Theor. Comput. Sci. **971**, 114063 (2023). https://doi.org/10.1016/J.TCS.2023.114063

[10] Bianchini, R., Dagnino, F., Giannini, P., Zucca, E.: Multi-graded Featherweight Java. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States. LIPIcs, vol. 263, pp. 3:1–3:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ECOOP.2023.3

[11] Bierman, G.M., De Paiva, V.C.: Intuitionistic necessity revisited. School of Computer Science research reports-University of Birmingham CSR (1996)

[12] Bierman, G.M., de Paiva, V.: On an intuitionistic modal logic. Stud Logica **65**(3), 383–416 (2000). https://doi.org/10.1023/A:1005291931660

[13] Brady, E.C.: Idris 2: Quantitative type theory in practice. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference). LIPIcs, vol. 194, pp. 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPICS.ECOOP.2021.9

[14] Brookes, S., Geva, S.: Computational comonads and intensional semantics. Carnegie-Mellon University. Department of Computer Science (1991)

[15] Brunel, A., Gaboardi, M., Mazza, D., Zdancewic, S.: A core quantitative coeffect calculus. In: Shao, Z. (ed.) Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8410, pp. 351–370. Springer (2014). https://doi.org/10.1007/978-3-642-54833-8_19

[16] Buiras, P., Vytiniotis, D., Russo, A.: HLIO: mixing static and dynamic typing for information-flow control in haskell. In: Fisher, K., Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. pp. 289–301. ACM (2015). https://doi.org/10.1145/2784731.2784758

[17] Choudhury, P.: Monadic and comonadic aspects of dependency analysis. Proc. ACM Program. Lang. **6**(OOPSLA2), 1320–1348 (2022). https://doi.org/10.1145/3563335, https://doi.org/10.1145/3563335

[18] Choudhury, P., III, H.E., Eisenberg, R.A., Weirich, S.: A graded dependent type system with a usage-aware semantics. Proc. ACM Program. Lang. **5**(POPL), 1–32 (2021). https://doi.org/10.1145/3434331

[19] Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976). https://doi.org/10.1145/360051.360056

[20] Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A., Pierce, B.C.: Linear dependent types for differential privacy. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 357–370. ACM (2013). https://doi.org/10.1145/2429069.2429113

[21] Gaboardi, M., Katsumata, S., Orchard, D.A., Breuvart, F., Uustalu, T.: Combining effects and coeffects via grading. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016. pp. 476–489. ACM (2016). https://doi.org/10.1145/2951913.2951939

[22] Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program. **20**(1), 19–50 (2010). https://doi.org/10.1017/S0956796809990268

[23] Ghica, D.R., Smith, A.I.: Bounded Linear Types in a Resource Semiring. In: Shao, Z. (ed.) Programming Languages and Systems - 23rd European

Symposium on Programming, ESOP 2014. Lecture Notes in Computer Science, vol. 8410, pp. 331–350. Springer (2014). https://doi.org/10.1007/978-3-642-54833-8_18

[24] Gifford, D.K., Lucassen, J.M.: Integrating functional and imperative programming. In: Scherlis, W.L., Williams, J.H., Gabriel, R.P. (eds.) Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6, 1986. pp. 28–38. ACM (1986). https://doi.org/10.1145/319838.319848

[25] Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1 – 101 (1987). https://doi.org/https://doi.org/10.1016/0304-3975(87)90045-4

[26] Girard, J., Scedrov, A., Scott, P.J.: Bounded linear logic: A modular approach to polynomial-time computability. Theor. Comput. Sci. **97**(1), 1–66 (1992). https://doi.org/10.1016/0304-3975(92)90386-T

[27] Hainry, E., Péchoux, R.: A type-based complexity analysis of object oriented programs. Inf. Comput. **261**, 78–115 (2018). https://doi.org/10.1016/J.IC.2018.05.006

[28] Hedin, D., Sabelfeld, A.: A perspective on information-flow control. In: Nipkow, T., Grumberg, O., Hauptmann, B. (eds.) Software Safety and Security - Tools for Analysis and Verification, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 33, pp. 319–347. IOS Press (2012). https://doi.org/10.3233/978-1-61499-028-4-319

[29] Heintze, N., Riecke, J.G.: The slam calculus: Programming with secrecy and integrity. In: MacQueen, D.B., Cardelli, L. (eds.) POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998. pp. 365–377. ACM (1998). https://doi.org/10.1145/268946.268976

[30] Hughes, J., Orchard, D.: Resourceful Program Synthesis from Graded Linear Types. In: Fernández, M. (ed.) Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12561, pp. 151–170. Springer (2020). https://doi.org/10.1007/978-3-030-68446-4_8

[31] Hughes, J., Orchard, D.: Program synthesis from graded types. In: Weirich, S. (ed.) Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14576, pp. 83–112. Springer (2024). https://doi.org/10.1007/978-3-031-57262-3_4

[32] Hughes, J., Vollmer, M., Orchard, D.: Deriving distributive laws for graded linear types. In: Lago, U.D., de Paiva, V. (eds.) Proceedings Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity&TLLA@IJCAR-FSCD 2020, Online, 29-30 June 2020. EPTCS, vol. 353, pp. 109–131 (2020). https://doi.org/10.4204/EPTCS.353.6

[33] Katsumata, S.: Parametric effect monads and semantics of effect systems. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 633–646. ACM (2014). https://doi.org/10.1145/2535838.2535846

[34] Lago, U.D., Gaboardi, M.: Linear Dependent Types and Relative Completeness. Log. Methods Comput. Sci. **8**(4) (2011). https://doi.org/10.2168/LMCS-8(4:11)2012

[35] Liepelt, V.B., Orchard, D., Marshall, D.: Same coeffect, different base: Graded types with and without linearity (2024), under review

[36] Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Ferrante, J., Mager, P. (eds.) Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988. pp. 47–57. ACM Press (1988). https://doi.org/10.1145/73560.73564

[37] Marshall, D., Orchard, D.: Graded Modal Types for Integrity and Confidentiality. In: 17th Workshop on Programming Languages and Analysis for Security (PLAS 2022) (2022), https://arxiv.org/abs/2309.04324

[38] Marshall, D., Orchard, D.: Functional Ownership through Fractional Uniqueness. Proc. ACM Program. Lang. **8**(OOPSLA1), 1040–1070 (2024). https://doi.org/10.1145/3649848, https://doi.org/10.1145/3649848

[39] Marshall, D., Orchard, D.: Non-linear communication via graded modal session types. Inf. Comput. **301**, 105234 (2024). https://doi.org/10.1016/J.IC.2024.105234

[40] Marshall, D., Vollmer, M., Orchard, D.: Linearity and Uniqueness: An Entente Cordiale. In: Sergey, I. (ed.) Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13240, pp. 346–375. Springer (2022). https://doi.org/10.1007/978-3-030-99336-8_13

[41] McBride, C.: I Got Plenty o' Nuttin', Lecture Notes in Computer Science, vol. 9600, pp. 207–233. Springer (2016). https://doi.org/10.1007/978-3-319-30936-1_12

[42] Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991). https://doi.org/10.1016/0890-5401(91)90052-4, https://doi.org/10.1016/0890-5401(91)90052-4

[43] Moon, B., III, H.E., Orchard, D.: Graded modal dependent type theory. In: Yoshida, N. (ed.) Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12648, pp. 462–490. Springer (2021). https://doi.org/10.1007/978-3-030-72019-3_17

[44] Mycroft, A., Orchard, D.A., Petricek, T.: Effect Systems Revisited - Control-Flow Algebra and Semantics. In: Probst, C.W., Hankin, C., Hansen, R.R. (eds.) Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays. Lecture Notes in Computer Science, vol. 9560, pp. 1–32. Springer (2016). https://doi.org/10.1007/978-3-319-27810-0_1

[45] Myers, A.C.: Jflow: Practical mostly-static information flow control. In: Appel, A.W., Aiken, A. (eds.) POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999. pp. 228–241. ACM (1999). https://doi.org/10.1145/292540.292561

[46] Orchard, D., Liepelt, V., III, H.E.: Quantitative program reasoning with graded modal types. PACMPL **3**(ICFP), 110:1–110:30 (2019). https://doi.org/10.1145/3341714, https://doi.org/10.1145/3341714

[47] Orchard, D.A.: Programming contextual computations. Tech. rep., University of Cambridge (2014), https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.708000

[48] Orchard, D.A., Bolingbroke, M., Mycroft, A.: Ypnos: declarative, parallel structured grid programming. In: Petersen, L., Pontelli, E. (eds.) Proceedings of the POPL 2010 Workshop on Declarative Aspects of Multicore Programming, DAMP 2010, Madrid, Spain, January 19, 2010. pp. 15–24. ACM (2010). https://doi.org/10.1145/1708046.1708053

[49] Orchard, D.A., Mycroft, A.: A Notation for Comonads. In: Hinze, R. (ed.) Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8241, pp. 1–17. Springer (2012). https://doi.org/10.1007/978-3-642-41582-1_1

[50] Orchard, D.A., Petricek, T., Mycroft, A.: The semantic marriage of monads and effects. CoRR **abs/1401.5391** (2014), http://arxiv.org/abs/1401.5391

[51] Petricek, T.: Context-aware programming languages. Ph.D. thesis, University of Cambridge (3 2017)

[52] Petricek, T., Orchard, D.A., Mycroft, A.: Coeffects: Unified static analysis of context-dependence. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M.Z., Peleg, D. (eds.) Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II. Lecture Notes in Computer Science, vol. 7966, pp. 385–397. Springer (2013). https://doi.org/10.1007/978-3-642-39212-2_35

[53] Petricek, T., Orchard, D.A., Mycroft, A.: Coeffects: a calculus of context-dependent computation. In: Jeuring, J., Chakravarty, M.M.T. (eds.) Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014. pp. 123–135. ACM (2014). https://doi.org/10.1145/2628136.2628160

[54] Pottier, F., Simonet, V.: Information flow inference for ML. ACM Trans. Program. Lang. Syst. **25**(1), 117–158 (2003). https://doi.org/10.1145/596980.596983

[55] Rajani, V., Garg, D.: Types for information flow control: Labeling granularity and semantic models. In: 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018. pp. 233–246. IEEE Computer Society (2018). https://doi.org/10.1109/CSF.2018.00024

[56] Rajani, V., Garg, D.: On the expressiveness and semantics of information flow types. J. Comput. Secur. **28**(1), 129–156 (2020). https://doi.org/10.3233/JCS-191382, https://doi.org/10.3233/JCS-191382

[57] Russo, A.: Functional pearl: two can keep a secret, if one of them uses haskell. In: Fisher, K., Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. pp. 280–288. ACM (2015). https://doi.org/10.1145/2784731.2784756

[58] Simonet, V., Rocquencourt, I.: Flow caml in a nutshell. In: Proceedings of the first APPSEM-II workshop. pp. 152–165 (2003)

[59] Snider, D.e.a.: HHVM and Hack User Documentation, https://web.archive.org/web/20240824005941/https://docs.hhvm.com/hack/contexts-and-capabilities/introduction

[60] Uustalu, T., Vene, V.: The essence of dataflow programming. In: Yi, K. (ed.) Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3780, pp. 2–18. Springer (2005). https://doi.org/10.1007/11575467_2

[61] Uustalu, T., Vene, V.: Signals and comonads. J. Univers. Comput. Sci. **11**(7), 1310–1326 (2005). https://doi.org/10.3217/JUCS-011-07-1311

[62] Uustalu, T., Vene, V.: Comonadic notions of computation. In: Adámek, J., Kupke, C. (eds.) Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science, CMCS 2008, Budapest, Hungary, April 4-6, 2008. Electronic Notes in Theoretical Computer Science, vol. 203, pp. 263–284. Elsevier (2008). https://doi.org/10.1016/J.ENTCS.2008.05.029

[63] Vollmer, V., Marshall, D., Eades III, H., Orchard, D.: A Mixed Linear and Graded Logic: Proofs, Terms and Models. In: 33rd EACSL Annual Conference on Computer Science Logic (CSL 2025) (2025). https://doi.org/10.4230/LIPIcs.CSL.2025.32, https://doi.org/10.4230/LIPIcs.CSL.2025.32

[64] Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996). https://doi.org/10.3233/JCS-1996-42-304

[65] Volpano, D.M., Smith, G.: Verifying secrets and relative secrecy. In: POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000. pp. 268–276 (2000). https://doi.org/10.1145/325694.325729, https://doi.org/10.1145/325694.325729

[66] Wadler, P.: Linear types can change the world! In: Broy, M., Jones, C.B. (eds.) Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990. p. 561. North-Holland (1990)
[67] Wadler, P., Thiemann, P.: The marriage of effects and monads. ACM Trans. Comput. Log. **4**(1), 1–32 (2003). https://doi.org/10.1145/601775.601776
[68] Walker, D.: Substructural type systems. Advanced topics in types and programming languages pp. 3–44 (2005)
[69] Wood, J., Atkey, R.: A linear algebra approach to linear metatheory **353**, 195–212 (2020). https://doi.org/10.4204/EPTCS.353.10