# DEPENDENCY AND LINEARITY ANALYSES IN PURE TYPE SYSTEMS

Pritam Choudhury

A DISSERTATION

in

Computer and Information Science (CIS)

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2022

Supervisor of Dissertation

Stephanie Weirich, ENIAC President's Distinguished Professor, CIS

Graduate Group Chairperson

Mayur Naik, Professor, CIS

Dissertation Committee

Rajeev Alur, Zisman Family Professor, CIS
Dominic Orchard, Senior Lecturer, School of Computing, University of Kent
Benjamin C. Pierce, Henry Salvatori Professor, CIS
Steve Zdancewic, Schlein Family President's Distinguished Professor, CIS

Dedicated to my Gurudeva, Śrī Śrī Ṭhākura, who showed me the way.

# Acknowledgement

One fine day, when I left my home in India for my doctoral studies at Penn, I didn't know how things would turn out. There was expectation, excitement, fear, hope, doubt, anticipation, all playing out in my heart. In the midst of this whirlpool of emotions, the faith of my mother and the spirit of my father kept me grounded. Standing upon these two pillars, I dreamed of creating something new. And thus, began my journey.

Six years have passed since then. During these six years, a lot has happened. I went through a diverse range of experiences and have learned from them. The successes and setbacks I had have together made me more resilient. The appreciations and criticisms I received have together made me more humble. The good and the bad days I had have together made me more mature. When I look back now, I see that my journey has been quite different from what I had expected. But nevertheless, I am happy at how it has turned out. I shall take this opportunity to acknowledge the people without whose participation this journey would not have been possible.

First, I would like to thank my advisor, Stephanie Weirich, who helped me develop my research ideas and supported me for the past several years. I appreciate the freedom she gave me in working on problems I found interesting. Second, I would like to thank my research collaborators, Harley Eades III, Richard Eisenberg, Antoine Voizard and Stephanie Weirich, with whom I worked on projects that have led to this dissertation. Third, I would like to thank the members of my Dissertation Committee, Rajeev Alur, Dominic Orchard, Benjamin C. Pierce and Steve Zdancewic, for their comments and feedback on my thesis proposal.

Next, I would like to thank my friends from the Penn PL Club, with whom I had several stimulating conversations on a wide range of topics. I would also like to thank other friends of mine, both here and back home, who lightened up my mood whenever we met. Thanks are also due to my younger sister who has been a very good friend all along this journey, cheering me up with her kind words of support and encouragement.

I shall also take this opportunity to thank two of my teachers from school days, Krishna Debnath and Keshab Chandra Saha, who taught me Maths with utmost care and instilled in me a love for the subject. That love for Maths motivated me to study Computer Science. So, I owe this thesis, in part, to their active encouragement during those formative years.

Finally, I would like to thank Master Felice Macera and other instructors of the Penn TaeKwonDo Club. Joining this club has been one of the best decisions of my life and I shall remain grateful to Master Macera for the kindness he has shown towards me in teaching martial arts.

# Abstract

Dependency analysis is necessary to control flow of information in programming languages. Linearity analysis is necessary to control usage of resources in programming languages. These analyses have a wide variety of applications. For example, analyzing binding-times of parts of programs, restricting variable use in programs, etc. What connects these analyses is that both of them need to model two different worlds (for ex., low and high security/linear and non-linear) with constrained mutual interaction. The constraint in dependency analysis is that information from the high-security world cannot leak into the low-security world. The constraint in linearity analysis is that derivations in the non-linear world cannot use assumptions from the linear world. Dependency and linear type systems statically ensure that interactions between worlds honor respective constraints. Several calculi have been proposed in literature for dependency and linearity analyses in simple and polymorphic type systems. However, with regard to dependent type systems, these analyses have not been explored much.

But with the growing use of dependent types both in theory and practice, dependency and linearity analyses in such type systems are much needed. In this dissertation, we address this need by generalizing dependency and linearity analyses to Pure Type Systems, which include several well-known dependent type systems. We build upon existing work on graded type systems to develop three specific calculi for Pure Type Systems: DDC for dependency analysis, GRAD for linearity analysis and LDC for combined dependency and linearity analysis. Our thesis is that these calculi provide a systematic way for analyzing dependency, linearity or a combination of the two in any Pure Type System. We study the metatheoretic properties of these calculi, show soundness of their analyses, discuss their applications and position them in the milieu of other dependency and linear calculi. Overall, our work provides insight into several nuances, hitherto unknown, of dependency and linearity analyses in both simple and dependent type systems.

# Contents

# Preface

Perspective plays an important role in shaping our understanding of the physical world. The same entity, when viewed from a different perspective, may appear to be very different. The ancient Indian story of the blind men and the elephant is an age-old testimony to this phenomenon. What makes this story interesting is that though each blind man had a unique description of the animal, all of them were equally correct. So in a hypothetical world inhabited solely by these blind men, the question, 'How does an elephant look like?', has multiple correct answers. The answers depend not just on the elephant but also on the men who came in contact with the animal. This story illustrates how our understanding of a physical entity is not just a function of the entity alone but also of factors external to the entity, in particular, our perspective.

Perspective plays an equally important role in academic disciplines like computer science, and in particular, in our area of interest: theory of programming languages. To provide a concrete instance, consider the question: Are two given programs equivalent? For some given pair of programs, the answer to this question can be both yes and no. For example, from a security perspective, suppose, two given programs output high-security values of the same type. Then, to an observer with just low-security clearance, these programs would appear equivalent while an observer with high-security clearance may find them to be different. So the equivalence of two programs is not just a function of the programs themselves but also of factors external to the programs, in particular, the 'perspective of the observer'.

Similar to perspective, environment also plays an important role in shaping our understanding of the physical world. The same entity, when viewed in a different environment, may appear to be very different. There is an ancient Indian story, similar to the above one, that illustrates this point: Four brothers seriously disagree in their descriptions of Kiṃśuka (*Butea monosperma*) tree because they saw the tree during different seasons. So the question, 'How does a Kiṃśuka tree look like?' has multiple correct answers, depending upon the season one sees the tree.

Environment plays an equally important role in the theory of programming languages. To provide a concrete instance, consider the question: Can an array be destructively updated? For some array, the answer to this question can be both yes and no. If the array has just one pointer to it, then it can be destructively updated. Otherwise, it should not be destructively updated. Thus, the choice of destructive update of an array is a function of the environment in which it exists, or more precisely, the number of pointers pointing at it.

'Perspective' and 'Environment' are notions central to dependency and linearity analyses respectively. This dissertation formalizes these notions towards analyzing dependency and linearity in pure type systems. This formalization answers important technical questions, bridges gaps between concepts and opens up new areas of exploration. Hopefully, this formalization will be useful to researchers working in programming languages, particularly in the theory and design of pure type systems.

# Chapter 1

# Introduction

Type systems formalize our intuition of correct programs: a correct program is one that is well-typed. Our intuition of correctness, however, varies depending upon application. For example: in a security system, correctness would entail absence of read access to secret files by public users; in a distributed system, correctness would entail absence of simultaneous write access to a file by multiple users. To formalize such notions of correctness, we employ dependency type systems and linear type systems. These type systems have a wide variety of applications.

Dependency type systems are good at *controlling information flow*. In the form of security type systems [Heintze and Riecke, 1998, Volpano et al., 1996], they guarantee that low-security outputs do not depend upon high-security inputs. In the form of binding-time type systems [Davies, 2017, Gomard and Jones, 1991], they guarantee that early-bound expressions do not depend upon late-bound ones. Dependency type systems are commonly used in practice, for example, in metaprogramming languages like MetaOcaml [Calcagno et al., 2003], in static analyzers like Jif extension of Java [Myers, 1999], etc.

Linear type systems are good at *managing resource usage and sharing*. In the form of session types, they provide useful guarantees like absence of deadlocks in distributed systems where resources like files, channels, etc. are shared among processes [Caires et al., 2016]. In functional programs, linear types can reason about state and enable compiler optimization like in-place update of memory locations [Wadler, 1990]. Owing to their utility, linear types have found their way into several programming languages like Haskell [Bernardy et al., 2018], Granule [Orchard et al., 2019], Idris 2 [Brady, 2021], etc.

Though dependency type systems and linear type systems serve different purposes, they essentially address the same abstract problem. The problem is to model two different worlds that interact following given constraints. Dependency type systems need to model low and high security worlds with the constraint that information from the high-security world cannot leak into the low-security world. Linear type systems need to model linear and non-linear worlds with the

constraint that derivations in the non-linear world cannot depend upon assumptions from the linear world. This fundamental similarity connects dependency and linearity analyses and forms the basis of this dissertation.

There are several type systems for dependency analysis [Abadi, 2006, Abadi et al., 1999, Davies, 2017, Hatcliff and Danvy, 1997, Heintze and Riecke, 1998, Shikuma and Igarashi, 2006, Volpano et al., 1996, etc.] and linearity analysis [Abramsky, 1993, Barber, 1996, Benton, 1994, Benton et al., 1993, Brunel et al., 2014, Ghica and Smith, 2014, etc.] in simple and polymorphic type systems. However, with regard to dependent type systems, these analyses have not been explored much. But with the growing use of dependent types both in theory and practice, dependency and linearity analyses in such type systems is much needed. The reason behind this need is that the problems these analyses address in simple and polymorphic type systems also arise in dependent type systems. For example, analyzing information flow, analyzing binding-time, reasoning about state, etc. are important in dependent type systems as well.

To address this need, we extend dependency and linearity analyses to Pure Type Systems. Recall that Pure Type System (PTS) [Barendregt, 1993] is a general formalism that captures many well-known dependent type systems like Calculus of Constructions, Type-in-Type, etc (in addition to capturing simple and polymorphic type systems like Simply-Typed $\lambda$-calculus, Polymorphic $\lambda$-calculus, etc). In this dissertation, we design the following calculi for Pure Type Systems: DDC for dependency analysis, GraD for linearity analysis and LDC for combined dependency and linearity analysis. Our calculi can control flow of information and manage usage of resources in dependently-typed programs. We use DDC for enforcing security and binding-time constraints in dependently-typed programs. We use GraD for enforcing constraints on variable use and reasoning about state in dependently-typed programs. LDC brings DDC and GraD under the same umbrella (roughly speaking) and combines their individual analyses.

The extension of dependency and linearity analyses to dependent type systems is a challenging problem. The main challenge lies in extending the analyses from just terms in simple/polymorphic type systems to both types and terms in dependent type systems. The difficulty of the challenge, particularly in relation to linearity analysis, is well-known [McBride, 2016]. With regard to dependency analysis, the problem did not receive much attention, partly because there are some calculi [Abel and Scherer, 2012, Bernardy and Guilhem, 2013, Mishra-Linger and Sheard, 2008, etc.] for analyzing specific dependencies in dependent type systems. Our interest, however, lies in a general dependency calculus for dependent type systems, one that can be used for a variety of analyses like security, binding-time, etc. The problem of designing such a calculus has been open.

Our work draws upon existing literature on linearity and dependency analyses. So, to appreciate the novelty of our work, it is important to understand it in the context of existing literature. As such, we briefly review the milieu of linearity and dependency analyses with special emphasis on type systems that acted as precursors to our work. We start with linearity analysis because it has a richer literature than dependency analysis.

## 1.1 Contextualizing Our Work on Linearity Analysis

### 1.1.1 Linear Logic

Linearity analysis is based on linear logic [Girard, 1987]. Linear logic is a logic of state [Girard, 1995]. Unlike traditional classical or intuitionistic logic where true propositions are always true, in linear logic, true propositions may change their state and become false. This flexibility enables linear logic reason about dynamic practical situations that are not readily captured by traditional logics. For example, consider the situation: Whenever I move out of my office, the statement 'I am in my office' changes its truth value. Traditional logics do not provide a straightforward way to reason about such dynamic statements. But in linear logic, we may model the situation with the axiom, $\texttt{moveOut} : \forall x.\texttt{in}(x) \multimap \texttt{out}(x)$, which means that $\texttt{moveOut}(x)$ is a transformation that consumes proposition $\texttt{in}(x)$ and produces proposition $\texttt{out}(x)$ ($x$ here stands for individuals). This idea of treating *propositions as resources* that are consumed and produced forms the basis of reasoning about state in linear logic.

If propositions are resources, then they should not be arbitrarily copied and discarded because otherwise, they lose their meaning. For example, consider the situation: I am in my office and I move out of my office. This situation may be represented as: $\texttt{in}(x) \otimes \texttt{moveOut}(x)$. If copying were allowed: $\texttt{in}(x) \otimes \texttt{moveOut}(x) \multimap \texttt{in}(x) \otimes \texttt{in}(x) \otimes \texttt{moveOut}(x) \multimap \texttt{in}(x) \otimes \texttt{out}(x)$, a contradiction. Thus, copying and discarding of propositions need to restricted. Linear logic does so by forgoing the structural rules of contraction and weakening respectively. So in linear logic, $\texttt{in}(x) \multimap \texttt{in}(x) \otimes \texttt{in}(x)$ is not true. While this change is necessary for reasoning about resources, it also limits the expressive power of the logic. To elaborate, from a resource perspective, the intuitionistic implication, $f : A \to B$, may be read as: $f$ consumes some number of copies of $A$ and produces one copy of $B$. This implication cannot be expressed in a straightforward manner unless copying and discarding of resources are allowed. To resolve this impasse, linear logic introduces the modality, !, also called an exponential. Resources under this modality may be copied and discarded without restriction. Such resources are *nonlinear*, as opposed to *linear* resources that may not be copied or discarded. To give an example, proposition $A$ in the intuitionistic implication $A \to B$ is a nonlinear resource whereas $B$ is a linear resource. In linear logic, the implication may be expressed as: $!A \multimap B$. With the !-modality, linear logic regains its expressive power; as a matter of fact, it subsumes standard intuitionistic logic. Thus, linear logic can be seen as an intuitionistic logic that can reason about state.

Now, programming languages need an intuitionistic logic [Martin-Löf, 1982] and the ability to reason about state. Linear logic can meet both these needs. So it could be a basis for new programming languages. This was realized soon enough: several linear type systems [Abramsky, 1993, Lafont, 1988, Lincoln and Mitchell, 1992, Wadler, 1990, 1994, etc.] immediately followed the introduction of linear logic.

Three decades have passed since then. During these decades, a lot of research has happened in the arena of linear type systems. Thanks to this research, linear types and their variants have found their way into mainstream programming languages like Haskell, Rust, etc. But the research on linear type systems continues unabated [Atkey, 2018, Choudhury et al., 2021, McBride, 2016, Moon et al., 2021, etc]. To understand where our work stands in the midst of all this research, we first need to draw an outline of the trajectory of the research itself.

### 1.1.2 Evolution of Linear Simple Type Systems

The problem of designing a linear type system, even for simple types, is far from trivial. Only with trial and error, did researchers come up with systems that are now regarded as standards. These standard systems may be easy to understand and work with but they were certainly not easy to come up with. So, we would like to caution the reader against hindsight bias as we discuss the evolution of linear type systems.

Linear type systems need to model two types of resources: linear resources that must be used exactly once and nonlinear resources that may be used any number (including 0) of times. How can the same type system model both linear and nonlinear resources? This question perplexed early researchers on linear type systems.

One answer, proposed by Abramsky [1993], was to let all resources (i.e. propositions) be linear by default and then mark the nonlinear ones among them with a type former, !. To illustrate, the resource $x : A$ is linear by default; but if $A = !B$, then it may be used as a nonlinear resource. Several early linear type systems [Benton et al., 1993, Lincoln and Mitchell, 1992, Wadler, 1994, etc.] followed this approach. While this approach works, it also has some issues. First, this approach leads to a problematic promotion rule. Recall the promotion rule: if every assumption in the derivation of a term is nonlinear, then the term itself may be used nonlinearly. In Abramsky's system, the promotion rule does not commute with substitution. As such, substitution becomes inadmissible, unless added as an explicit typing rule. Later authors, Benton et al. [1993], fixed this problem by baking substitution directly into the promotion rule. However, such a fix is not very satisfactory because it results in a complex promotion rule. Second, this approach leads to verbose type systems. Such type systems require explicit term-level constructs (and therefore typing rules) for promotion, dereliction, contraction and weakening. Recall that dereliction enables linear use of a nonlinear resource while contraction and weakening enable copying and discarding of nonlinear resources respectively. Promotion may be thought of as an introduction form for ! with dereliction, contraction and weakening as its elimination forms. Having three elimination forms for a type former leads to an involved semantics. With these issues around, the search for better linear type systems was on.

On a closer analysis, one finds that the root cause of the issues described above is the safe but restrictive assumption that all resources are linear by default. The assumption is safe because nonlinear resources may be treated linearly. But the assumption is also restrictive because

nonlinear resources need to be identified and treated differently. A question then arises: In lieu of this round about way of tracking resources, why not distinguish linear and nonlinear resources from the very outset? This insight laid the foundation for the later and now standard linear simple type systems, the Linear Nonlinear (LNL) $\lambda$-calculus of Benton [1994] and the Dual Intuitionistic Linear Logic (DILL) of Barber [1996].

Both Benton [1994] and Barber [1996] split contextual assumptions into two zones: nonlinear and linear. Assumptions that are in the nonlinear zone may be used without restriction. However, assumptions that are in the linear zone must be used linearly. With this simple modification, Benton [1994] and Barber [1996] avoided the issues their predecessors faced. To elaborate, the promotion rule in their systems is quite straightforward: promote whenever the linear zone in the context is empty. Their systems do not have explicit constructs for weakening and contraction: they are carried out implicitly in the nonlinear zone. In short, LNL $\lambda$-calculus and DILL provided simple and clean formalism, something that was missing from the earlier systems.

### 1.1.3 Evolution of Linear Dependent Type Systems

LNL $\lambda$-calculus and DILL are effective in tracking linearity in simple type systems. So in due course, these systems were extended to dependent types. Krishnaswami et al. [2015] extended LNL $\lambda$-calculus to dependent types while Vákár [2015] did the same for DILL. The systems of Krishnaswami et al. [2015] and Vákár [2015] are interesting because they bring linearity and dependent types closer. However, they do so only after imposing a major restriction: *types cannot depend upon linear assumptions*. One might say, with this restriction in place, these systems are not really combining linearity with dependent types. For example, consider the following functions from Agda Standard Library [Agda Team, 2021], with their types refined: `fromℕ : (n : ℕ) ⊸ Fin (suc n)` and `_ℕ-ℕ_ : (n : ℕ) ⊸ Fin (suc n) ⊸ ℕ`. Here, $\mathbb{N}$ is the type of natural numbers and `Fin n` is the type of finite numbers less than `n`. The function `fromℕ` takes a natural number and returns the 'same' number. The function `_ℕ-ℕ_` takes a natural number `n` (minuend) and a finite number less than or equal to `n` (subtrahend) and returns the difference between the two. The types of these functions cannot be expressed in the systems under discussion because the types depend upon the linear assumption, `n : ℕ`. Thus, these systems leave something to be desired in their combination of linearity and dependent types.

If we carefully look at the systems of Krishnaswami et al. [2015] and Vákár [2015], we realize that the restriction they place on types is not their choice per se but is forced upon them by the simple type systems they extend. To see why, consider the application rule of DILL (rule DILL_App) and a hypothetical extension (rule DILL_DApp) of this rule to dependent types:

$$\textsc{DILL-App}\ \frac{\Delta\ ;\ \Gamma_1 \vdash b : A \multimap B \qquad \Delta\ ;\ \Gamma_2 \vdash a : A}{\Delta\ ;\ \Gamma_1, \Gamma_2 \vdash b\ a : B} \qquad \textsc{DILL-DApp}\ \frac{\Delta\ ;\ \Gamma_1 \vdash b : (x : A) \multimap B \qquad \Delta\ ;\ \Gamma_2 \vdash a : A}{\Delta\ ;\ \Gamma_1, \Gamma_2 \vdash b\ a : B\{a/x\}}$$

Here $\Delta$ and $\Gamma$ denote the nonlinear and linear zones of the context respectively. Observe that in rule DILL_DApp, the type $A$ can depend neither upon $\Gamma_1$ nor upon $\Gamma_2$ because $\Delta$, $\Gamma_1$ and $\Gamma_2$ are mutually disjoint. As such, $A$ can depend only upon $\Delta$. In other words, $A$ cannot depend upon any linear assumption.

Though rule DILL_App works well in a simply-typed system, its extension, rule DILL_DApp, does not work well in a dependently-typed system because it forces types to not depend upon linear assumptions. This example suggests that for extending linearity analysis to dependent types, one needs to start out with a type system that is more malleable. The foundation for such type systems was already laid down by bounded linear logic.

### 1.1.4 Bounded Linear Logic

Linear logic allows linear and nonlinear use of resources. While linear use precisely means single use, nonlinear use may mean any number of uses. Bounded linear logic (BLL) [Girard et al., 1992] brings precision into nonlinear use by introducing fine-grained distinctions on such use. For example, use upper-bounded by $n$, for any $n \in \mathbb{N}$. To express fine-grained notions of nonlinear use, BLL grades the exponential modality, !, of linear logic. For example, a resource $A$ that may be used at most $n$ times is represented as $!_n A$. Such fine-grained notions of use enables resource arithmetic within the logic itself. For example: From the sequent $!_{n_1} A_1, !_{n_2} A_2 \vdash B$, one can infer $!_{n \cdot n_1} A_1, !_{n \cdot n_2} A_2 \vdash !_n B$. From the sequent $!_{n_1} A, !_{n_2} A \vdash B$, one can infer $!_{n_1 + n_2} A \vdash B$. Note here that BLL can also reason about no use through $!_0$-modality.

BLL [Girard et al., 1992], however, does not allow unrestricted use. Thus, even though BLL can express fine-grained notions of use, it does not subsume linear logic. In particular, BLL cannot model the !-modality of linear logic because there is no natural number $n$ such that $n = n + 1$. This, however, should not be seen as a shortcoming since BLL was designed to capture polynomial time computations and unfettered exponential modality can express exponential time computations. But if we move away from the domain of complexity analysis to the domain of programming languages where we are at ease even with nonterminating computations, we may wish to allow unrestricted use alongside bounded use. That would enable fine-grained reasoning of usage without compromising on the benefits accorded by linear logic. To this end, the infinite cardinal $\omega$, a solution of the equation $n = n + 1$, is introduced. (Strictly speaking, the symbol $\aleph_0$ should have been used here in lieu of $\omega$. But we go with $\omega$, following existing literature on linear type systems.) The exponential modality, !, of linear logic is then regarded as $!_\omega$.

BLL inspired many simple/polymorphic type systems [Brunel et al., 2014, Ghica and Smith, 2014, Petricek et al., 2014, etc.] which ultimately paved the path for better linear dependent type systems. Interestingly, the type systems inspired by BLL are all parametrized by abstract algebraic structures that model resource arithmetic. The idea of parametrization of the type system by an algebraic structure marked a key development in the evolution of linear type systems. So next, we motivate the algebraic structure that models resource arithmetic in type systems based on BLL.

### 1.1.5 An Algebraic Structure for Resource Arithmetic

Till now, we have used natural numbers (and $\omega$) to model resource arithmetic. But, in lieu, we could have used elements of an appropriate abstract algebraic structure, one that would support operations on resources, i.e. addition and multiplication, the way we understand these operations. Going through abstract algebra, we find that a semiring [Golan, 1999] is an appropriate structure for this purpose. It has two binary operators, + (addition) and $\cdot$ (multiplication), that obey standard associative and distributive axioms and have identity elements, 0 and 1 respectively. Further, 0 is an annihilator for $\cdot$ and + is commutative. To give examples: $\mathbb{N}$ and $\mathbb{N} \cup \{\omega\}$ and $\{0, \omega\}$, with addition and multiplication defined as usual, are semirings.

With a semiring $\mathcal{Q}$ as a *parameter* to the type system, the elements of $\mathcal{Q}$ can be used as *grades* to distinguish among various notions of use. For example, with $\mathcal{Q} \coloneqq \{0, \omega\}$, the modalities $!_0$ and $!_\omega$ can distinguish between no use and unrestricted use. So, if one wishes to track no use and unrestricted use only, one can use the semiring $\{0, \omega\}$ (note $1 \in \mathbb{N}$ is absent from this semiring) as a parameter. This example shows that parametrization makes usage tracking flexible.

Before moving any further, we need to clarify an issue. Linear logic [Girard, 1987] does not allow discarding of resources, unless they are under the exponential modality. BLL [Girard et al., 1992], on the other hand, allows arbitrary discarding of resources. So the modality $!_1$ of BLL corresponds to affine (at most linear) use and not to linear use. Since BLL was designed to reason about *bounded* use, this feature does not pose any problem in the logic. However, a side effect of this feature is that BLL can not reason about strict linear use. But in type systems, we may want to reason about both bounded and strict use.

To enable reasoning about both strict and bounded use, we include a notion of *order* in the parametrizing structure itself. We modify the parametrizing structure from a semiring to a preordered semiring. With this modification, the same type system can reason about both strict and bounded use. For strict use, the parameter is set to a semiring with discrete order; for bounded use, the parameter is set to a semiring with total order. Thus, parametrization makes type systems versatile in tracking usage.

A preordered semiring is an ideal algebraic structure to reason about use. Thus, it comes as no surprise that the type systems [Brunel et al., 2014, Ghica and Smith, 2014, Petricek et al., 2014, etc.] based on BLL are parametrized by preordered semirings or similar structures.

### 1.1.6 Type Systems Based on Bounded Linear Logic

The type systems [Brunel et al., 2014, Ghica and Smith, 2014, Petricek et al., 2014, etc.] based on BLL are parametrized by preordered semirings or similar algebraic structures. In these systems, the elements of the parametrizing structures are used to grade assumptions in contexts of typing judgments. Such graded contexts are then manipulated by the typing rules via the operations of the parametrizing structure. These context manipulations help track resource usage in these systems. In a way, these systems may be seen as generalization of the split-context systems of Benton [1994] and Barber [1996]. While the split-context systems allow only two zones in the context, these systems implicitly allow as many zones as the number of elements of the parametrizing structure.

However, the systems based on BLL are more malleable than the split-context systems. In split-context systems, an assumption cannot simultaneously appear in linear and nonlinear zones in different premise judgments of a typing rule. However, in systems based on BLL, an assumption can appear at different grades in different premise judgments of a typing rule. A comparison of the application rule makes this point clear:

$$\textsc{dill-App}\ \frac{\Delta\ ;\ \Gamma_1 \vdash b : A \multimap B \qquad \Delta\ ;\ \Gamma_2 \vdash a : A}{\Delta\ ;\ \Gamma_1, \Gamma_2 \vdash b\ a : B} \qquad \textsc{bll-App}\ \frac{\Gamma'_1 \vdash b : A \multimap B \qquad \Gamma'_2 \vdash a : A \qquad \lfloor \Gamma'_1 \rfloor = \lfloor \Gamma'_2 \rfloor}{\Gamma'_1 + \Gamma'_2 \vdash b\ a : B}$$

Note that in rule BLL_App, $\lfloor \Gamma' \rfloor$ denotes the context underlying $\Gamma'$ i.e. $\Gamma'$ without grades. Further, note that $\Gamma'_1 + \Gamma'_2$ denotes the context formed by pointwise addition of grades of $\Gamma'_1$ and $\Gamma'_2$. Now, in rule DILL_App, $\Delta, \Gamma_1$ and $\Gamma_2$ are mutually disjoint but in rule BLL_App, $\Gamma'_1$ and $\Gamma'_2$ have the same underlying set of assumptions, possibly held at different grades. So an assumption $z : C$ can appear as $z :^1 C$ and $z :^0 C$ in $\Gamma'_1$ and $\Gamma'_2$ respectively but $z : C$ cannot simultaneously appear in $\Gamma_1$ (or $\Gamma_2$) and $\Delta$.

This flexibility in the type systems based on BLL becomes particularly useful in a dependent setting. To see how, compare rule DILL_DApp (extension of rule DILL_App to dependent types) with rule BLL_DApp (extension of rule BLL_App to dependent types):

$$\textsc{dill-DApp}\ \frac{\Delta\ ;\ \Gamma_1 \vdash b : (x : A) \multimap B \qquad \Delta\ ;\ \Gamma_2 \vdash a : A}{\Delta\ ;\ \Gamma_1, \Gamma_2 \vdash b\ a : B\{a/x\}} \qquad \textsc{bll-DApp}\ \frac{\Gamma'_1 \vdash b : (x : A) \multimap B \qquad \Gamma'_2 \vdash a : A \qquad \lfloor \Gamma'_1 \rfloor = \lfloor \Gamma'_2 \rfloor}{\Gamma'_1 + \Gamma'_2 \vdash b\ a : B\{a/x\}}$$

As discussed before, in rule DILL_DApp, the type $A$ cannot depend upon $\Gamma_1$ or $\Gamma_2$. But in rule BLL_DApp, $A$ can depend upon $\Gamma'_1$ (or $\Gamma'_2$) because $\Gamma'_1$ and $\Gamma'_2$ contain the same underlying set of assumptions. This flexibility makes it possible for types to depend upon linear assumptions. For

instance, in rule BLL_DApp, $A$ can depend upon a linear assumption in $\Gamma_1'$, say $z :^1 C$, because $z : C$ also appears in $\Gamma_2'$, possibly at a different grade. However, in rule DILL_DApp, $A$ cannot depend upon any assumption in $\Gamma_1$ (or in $\Gamma_2$) because $\Delta$, $\Gamma_1$ and $\Gamma_2$ are mutually disjoint.

The type systems based on BLL show a way of overcoming the restriction put on types by the linear dependent systems of Krishnaswami et al. [2015] and Vákár [2015]. For combining linearity with dependent types, the type systems based on BLL are, therefore, better starting points than the split-context systems of Benton [1994] and Barber [1996]. McBride [2016] first realized this fact. As such, his work shaped the latest chapter in the development of linear dependent type systems.

### 1.1.7 State of the Art in Linear Dependent Type Systems

McBride [2016] realized that the type systems based on BLL can be extended to dependent types without requiring types to not depend upon linear assumptions. However, for a successful combination of linearity and dependent types, he needed to address another challenge. If both a type and a term can depend upon a linear resource, how is that linear resource being used only once? McBride's answer to this question is that usage in types is 'contemplative' whereas usage in terms is 'consumptive'. In other words, according to McBride, usage in types does not count as real usage, only usage in terms does. For example, the linear assumption $(x : A)$ in the type $(x : A) \multimap B$ may be used any number of times in $B$ (because such use is 'contemplative') but only once in functions having type $(x : A) \multimap B$ (because such use is 'consumptive'). With this distinction in place, McBride [2016] designed the first-of-its-kind linear dependent type system.

McBride's insights are brilliant. But unfortunately, his system does not admit substitution [Atkey, 2018]. In a way, his system is too general to admit substitution. Atkey [2018] restricted his system in several ways and proposed a modified system, QTT. QTT adopts McBride's doctrine of contemplative vs. consumptive use but also admits substitution. Atkey [2018] shows soundness of QTT via categorical models. However, an issue with QTT is that types and terms are treated very differently in the calculus: types live in a resource-agnostic world while terms live in a resource-aware world. This non-uniform treatment of types and terms makes QTT more complex than is necessary. This drawback led us to design an alternative system, GrAD [Choudhury et al., 2021], for combining linearity with dependent types. In GrAD, unlike QTT, both types and terms live in a resource-aware world, resulting in a simpler and more uniform system.

GrAD [Choudhury et al., 2021] is an extension of the simple/polymorphic type systems [Brunel et al., 2014, Ghica and Smith, 2014, Petricek et al., 2014] based on BLL to dependent types. Both GrAD and these type systems use the same form of typing judgment. But GrAD extends usage tracking from terms to both types and terms. With regard to usage tracking in types, GrAD takes a principle somewhat different from that of McBride [2016]. According to McBride [2016], usage in a type does not ever count. According to Choudhury et al. [2021], usage in a

type does not count whenever it appears as a type in a typing judgment but usage in a type does count whenever it appears as a term in a typing judgment. For example: usage in $A$ does not count in the judgment $\Gamma \vdash a : A$ but usage in $A$ does count in the judgment $\Gamma' \vdash A : B$. More precisely, given any typing judgment $\Gamma \vdash a : A$ in GrAD, $\Gamma$ denotes the resources used by $a$, irrespective of whether $a$ is a term or a type. This uniform treatment of terms and types with respect to resource usage results in a clean formalism in GrAD.

Yet another approach may be taken towards accounting usage in types. Moon et al. [2021] present a system, Grtt, where usage in types and term are simultaneously accounted in typing judgments. To provide an example, given a standard typing judgment $x_1 : A_1, x_2 : A_2, x_3 : A_3 \vdash b : B$, Grtt tracks: the usage of $x_1$ in $A_2$, $A_3$, $b$ and $B$; the usage of $x_2$ in $A_3$, $b$ and $B$; the usage of $x_3$ in $b$ and $B$. This elaborate tracking of usage helps in several analyses. For example: Simply-typed $\lambda$-calculus can be embedded into Grtt by disallowing (through grade 0) dependence of types on term variables. Tracking usage in types all throughout makes Grtt a powerful calculus but it also makes Grtt a complex calculus: the typing rules of Grtt are particularly intricate.

The calculi QTT [Atkey, 2018], GrAD [Choudhury et al., 2021] and Grtt [Moon et al., 2021] represent the state of the art in linear dependent type systems. Here, we have given an overview of how they relate to one another. This overview positions our work, GrAD, in the field of linear dependent type systems. This section, as a whole, positions our work, GrAD, in the broader field of linearity analysis. We present the details of GrAD in Chapter 4.

## 1.2 Contextualizing Our Work on Dependency Analysis

### 1.2.1 Dependency Analysis in Programming Languages

Dependency analysis is the analysis of dependence of an entity upon another. The entities are primarily programs or parts thereof, but they can also be abstract, like security clearance levels in an organization, stages in a compilation process, etc. Broadly speaking, an entity depends upon another one if the latter influences the behavior of the former. On the other hand, an entity is independent of another one if the latter does not interfere in the behavior of the former. For example, consider the following $\lambda$-terms: $(\lambda x.x)\,(2+2)$ and $(\lambda x.4)\,(2+2)$. The argument $(2+2)$ dictates the normal form of the first term whereas it plays no role in deciding the normal form of the second term. So, we say that the first term depends upon the argument whereas the second one does not. What this means is that in the second term, we can replace the argument $(2+2)$ with any other terminating computation, while maintaining the same normal form for the term as a whole.

The power of dependency analysis comes from this very simple principle: if an entity does not depend upon another one, then variations in the latter should not affect the former. This is

the well-known principle of *noninterference* [Goguen and Meseguer, 1982, Jones and Lipton, 1975]. This principle has far-reaching implications and lies at the heart of dependency analyses in programming languages, like secure information flow analysis, binding-time analysis, etc.

In secure information flow analysis [Heintze and Riecke, 1998, Smith and Volpano, 1998], one wishes to guarantee that there is no flow of information from high-security data to low-security variables. Viewed abstractly in terms of security levels, this is equivalent to saying that level **Low**, denoting low-security computations, *does not depend* upon level **High**, denoting high-security computations. In binding-time analysis [Davies, 2017, Hatcliff and Danvy, 1997], one wishes to guarantee that a given program can be correctly compiled in multiple stages even when each stage can potentially optimize based on inputs received from earlier stages. To ensure correctness of such compilation, it is necessary that level **Early**, denoting early-stage computations, *does not depend* upon level **Later**, denoting later-stage computations. Secure information flow analysis and binding-time analysis are two well-known examples of dependency analysis. There are many other examples of dependency analysis [Abadi et al., 1996, Palsberg and Ørbæk, 1995, Tang and Jouvelot, 1995, Tip, 1995, Tofte and Talpin, 1997, etc.] appearing in literature.

### 1.2.2 An Algebraic Structure for Dependency Constraints

The examples above discuss dependency analysis with respect to two levels: **Low** and **High**, **Early** and **Later**. Such an analysis can be extended to an arbitrary (finite) number of levels with dependency constraints among them. Denning [1976] observed that dependency constraints among levels result in a lattice structure [Birkhoff, 1967]. For example, consider the following set of security levels: a low-security level $\mathbf{L}$, two medium-security levels $\mathbf{M_1}$ and $\mathbf{M_2}$ that do not share information between each other, and a high-security level $\mathbf{H}$. These constraints can be modeled by a diamond lattice, $\mathcal{L}_\diamond$, where $\mathbf{L} \sqsubseteq \mathbf{M_1} \sqsubseteq \mathbf{H}$ and $\mathbf{L} \sqsubseteq \mathbf{M_2} \sqsubseteq \mathbf{H}$, but $\mathbf{M_1} <> \mathbf{M_2}$. Any information flow that goes against the order of this lattice model would be illegal.

The idea behind lattice model of dependency is that level $\ell_2$ can depend upon level $\ell_1$ if and only if $\ell_1 \sqsubseteq \ell_2$. The purpose of the lattice model is to provide an abstract structure to reason about dependency constraints. Dependency constraints among security levels or among stages of compilation can be represented through lattices. The lattices corresponding to the examples discussed would be: $\mathbf{Low} \sqsubset \mathbf{High}$, $\mathbf{Early} \sqsubset \mathbf{Later}$.

One can take a particular set of dependency constraints, represented as a lattice, and design a calculus that enforces these constraints. There are many calculi [Davies, 2017, Hatcliff and Danvy, 1997, Smith and Volpano, 1998, etc.] that analyze specific dependencies in this manner. However, our interest lies in a general calculus of dependency, one that can analyze a wide range of dependencies that appear in security, compilation, etc. Such a calculus would not be fine-tuned to a specific dependency lattice but would be able to reason about arbitrary dependency lattices. One such calculus and the first of its kind is the Dependency Core Calculus of Abadi et al. [1999].

### 1.2.3 Dependency Core Calculus

Over two decades ago, Abadi et al. [1999] showed that several dependency analyses [Abadi et al., 1996, Hatcliff and Danvy, 1997, Heintze and Riecke, 1998, Tang and Jouvelot, 1995, Volpano et al., 1996] can be seen as instances of a general Dependency Core Calculus (DCC). Their work has served as a foundational framework for dependency analysis in the field of programming languages and has led to extensive research [Algehed, 2018, Algehed and Bernardy, 2019, Bowman and Ahmed, 2015, Shikuma and Igarashi, 2006, Tse and Zdancewic, 2004, etc.] on this topic.

DCC is a simply-typed calculus that is parametrized by an abstract dependency lattice. DCC may be seen as a minor extension of Moggi's computational metalanguage [Moggi, 1991]. The computational metalanguage is a general calculus for analyzing computational effects like nontermination, exceptions, input/output, etc. What makes DCC special though is that it integrates the lattice model of dependency with the computational metalanguage by grading the *monadic modality*, $T$, of the latter with elements of the former. This integration enables DCC enforce dependency constraints, modeled by an arbitrary lattice, in its type system. For example: given lattice $\mathcal{L}_\diamond$, DCC forces any function of type $T_{\mathbf{H}}\,\mathbf{Bool} \to T_{\mathbf{L}}\,\mathbf{Bool}$ to be constant. Here, $T_{\mathbf{H}}\,\mathbf{Bool}$ and $T_{\mathbf{L}}\,\mathbf{Bool}$ denote high and low security booleans respectively. Note that a non-constant function of type $T_{\mathbf{H}}\,\mathbf{Bool} \to T_{\mathbf{L}}\,\mathbf{Bool}$ would leak information from high to low security world, thereby breaching security.

Over the years, DCC has become 'the calculus' for dependency analysis in simple type systems. However, in spite of its success, DCC has its limitations. First, the monadic bind rule of the calculus is nonstandard and relies upon an auxiliary protection judgment. Second, owing to the nonstandard bind rule, it is not clear how DCC can be extended to dependent types. Third, being of a monadic nature, the calculus cannot capture dependency analyses that possess a comonadic nature, for example, the binding-time calculus, $\lambda^\circ$, of Davies [2017].

Our goal in this dissertation is to design a general dependency calculus for pure type systems. To the best of our knowledge, no such calculus has been proposed in literature. Given that DCC is a general dependency calculus for simple type systems, we thought that we could extend DCC to dependent types. However, with the above limitations of DCC, we did not find a way to do so. This, in turn, motivated us to take a relook at DCC and address its limitations in the first place.

### 1.2.4 Our Calculi for Dependency Analysis

We address the above limitations of DCC by designing an alternative dependency calculus, $\mathrm{GMCC_e}$, that is inspired by standard ideas from category theory. $\mathrm{GMCC_e}$ is both monadic and comonadic in nature and subsumes both monadic DCC and comonadic $\lambda^\circ$. Our construction explains the nonstandard bind rule and the protection judgment of DCC in terms of standard categorical concepts. It also leads to a novel technique for proving correctness of dependency

analysis. We use this technique to present alternative proofs of correctness for DCC and $\lambda^\circ$. We present the details of $\text{GMCC}_\text{e}$ in Chapter 2.

$\text{GMCC}_\text{e}$ is a simply-typed dependency calculus. In Chapter 3, we extend $\text{GMCC}_\text{e}$ to pure type systems. We design two calculi, $\text{DDC}^\top$ and DDC, for dependency analysis in pure type systems. Both $\text{DDC}^\top$ and DDC can analyze dependencies, over an arbitrary lattice, in pure type systems. However, DDC is more general than $\text{DDC}^\top$ because not only does it analyze dependencies but also it internalizes dependency analysis in typing itself. In other words, if dependency analysis in DDC were incorrect, then typing itself would be unsound. In most dependency calculi, including $\text{DDC}^\top$, correctness of dependency analysis does not affect typing. Internalizing dependency analysis in typing has several benefits that we point out in Chapter 3.

A dependency analysis that is particularly useful in dependently-typed languages is irrelevance analysis. Irrelevance analysis (in dependently-typed languages) is a widely studied problem with rich literature. Several calculi [Abel and Scherer, 2012, Barras and Bernardo, 2008, Miquel, 2001, Mishra-Linger, 2008, Pfenning, 2001, Tejiščák, 2020, etc.] address this problem, in its various forms. However, these calculi do not draw the connection between a general dependency analysis and irrelevance analysis. In this dissertation, we establish this connection by employing our general dependency calculi, $\text{DDC}^\top$ and DDC, for analyzing irrelevance. We find that our calculi can analyze fine-grained notions of irrelevance and resolve some open problems in this area. Next, we review the problem of irrelevance analysis and corresponding literature with the aim of contextualizing our contributions in this area.

### 1.2.5 Irrelevance Analysis in Dependent Type Systems

Irrelevance analysis is very useful in dependently-typed languages, both in the context of type-checking and run-time evaluation.

First, we describe the utility of irrelevance analysis in the context of run-time evaluation. Significant parts of dependently-typed programs, necessary to satisfy the type-checker, may be irrelevant to evaluation of those programs. For example, proofs appearing in dependently-typed programs, while necessary for type-checking, are generally run-time irrelevant. As an instance, consider the following function from Agda Standard Library [Agda Team, 2021], `inject≤ : Fin m → m ≤ n → Fin n`. This function takes a finite number less than `m` and a proof that `m ≤ n` and returns back the 'same' finite number. For typing this function, it's necessary that `m` be less than `n`. But for evaluating this function, this information is irrelevant. So the proof of `m ≤ n` that appears in the definition of `inject≤` may be erased after type-checking and before evaluation. Such erasure results in better space-time characteristics during evaluation.

Next, we describe the utility of irrelevance analysis in the context of type-checking. Consider the following Agda function: `phantom = λ x . if (fib 28 == 317810) then Nat else Bool : Nat → Set`, that ignores its argument and just returns either `Nat` or `Bool` based on a conditional that checks whether the Fibonacci number $F_{28}$ equals 317810. Using `phantom`, we

define: `conv = λ y . y : phantom 0 → phantom 1`. The function `conv` is essentially an identity function. We can immediately see that `conv` should type-check: since `phantom` ignores its argument, `phantom m = phantom n`. But, without irrelevance analysis, the type-checker needs to reduce `phantom 0` and `phantom 1` to find out whether they are the same type. With a computationally intensive conditional, such a reduction may take a long time. Thus, irrelevance analysis may help improve space-time characteristics of type-checking.

Analysis of irrelevance with regard to type-checking and run-time evaluation are referred to as *compile-time irrelevance* and *run-time irrelevance* analyses respectively. Run-time irrelevance analysis identifies sub-expressions that do not affect the result of evaluation while compile-time irrelevance analysis identifies sub-expressions that are not needed for type-checking. Both run-time irrelevance and compile-time irrelevance analyses are dependency analyses that need to enforce the constraint that level **Relevant**, denoting relevant expressions, does not depend upon level **Irrelevant**, denoting irrelevant expressions. We present the details of these analyses in $\text{DDC}^{\top}$ and DDC in Chapter 3.

Analysis of run-time irrelevance and compile-time irrelevance is a well-studied problem in the design of dependent type systems. In some systems, the focus is only on support for run-time irrelevance: see Mishra-Linger and Sheard [2008], Tejiščák [2020]. In other systems, the focus is on compile-time irrelevance: see Abel and Scherer [2012], Pfenning [2001]. Some systems support both, but require them to overlap: see Barras and Bernardo [2008], Miquel [2001], Mishra-Linger [2008]. The system of Moon et al. [2021] we saw earlier (in Section 1.1.7) does not require them to overlap but does not also make use of compile-time irrelevance while checking for equality of types.

Our dependency calculus, DDC, is the only system we are aware of that analyzes run-time irrelevance and compile-time irrelevance separately and makes use of the latter while checking for equality of types. Further, DDC analyzes these irrelevances in the presence of strong Σ-types with erasable first components. To the best of our knowledge, no prior work has been able to model such Σ-types in a system that analyzes compile-time irrelevance. The exact nature of the problem posed by such Σ-types and the way we resolve it is discussed in detail in Chapter 3.

Now that we have contextualized our work on linearity and dependency analyses, we draw a comparison between the two.

## 1.3 Linearity and Dependency Analyses: A Comparison

In this dissertation, we design calculi for linearity and dependency analyses in pure type systems. One question that may be asked after such a design is whether these calculi can be unified into a single calculus. In Chapter 5, we answer this question in the affirmative. We design a unified calculus, LDC, for combined linearity and dependency analysis in pure type systems.

Unifying linearity and dependency analyses is a challenging problem. We are not aware of any system in literature that unifies these analyses, even in a simply-typed setting. To understand the challenges in unifying these analyses, we need to see how they are similar to and different from one another.

### 1.3.1 Preordered Semiring vs. Lattice

In the beginning of this chapter, we pointed out the fundamental similarity connecting dependency and linearity analyses: both of them need to model two different worlds that interact following given constraints. As we dived deeper into these analyses, we saw that they need not be restricted to model two worlds only but can be generalized to model an arbitrary number of worlds. We also saw that the constraints upon interactions among these worlds can be modeled by abstract algebraic structures. For linearity analysis, the algebraic structures are preordered semirings. For dependency analysis, the algebraic structures are lattices. These structures, while similar in some ways, are also different in other ways.

Both preordered semirings and lattices are algebraic structures with two binary operators and a binary order relation. However, one crucial distinction between these structures is that while semirings need to ensure that one operator (multiplication) distributes over the other (addition), lattices do not need to ensure any such property. As a matter of fact, there are lattices where either operator does not distribute over the other. The simplest of such lattices is $M_3$ [Birkhoff, 1967] that has 5 elements: $\bot, \ell_1, \ell_2, \ell_3, \top$, ordered as: $\bot \sqsubseteq l_i \sqsubseteq \top$, where $i = 1, 2, 3$. In $M_3$, join and meet do not distribute over one another. Thus, an arbitrary lattice can not be viewed as a preordered semiring. However, distributive lattices, i.e. lattices where join and meet distribute over one another, may be viewed as preordered semirings where multiplication, addition and order are given by join, meet and lattice order respectively. But we see no principled justification in restricting dependency lattices to distributive ones only.

Another crucial distinction between preordered semirings and lattices is that while lattice operations (join and meet) are idempotent, semiring operations (addition and multiplication) need not be so. As a matter of fact, in the preordered semirings of our interest, like the semiring of natural numbers with discrete/total order, the operations are not idempotent. So these semirings can not be viewed as lattices.

Thus, to unify linearity and dependency analyses, we cannot use either of the algebraic structures as the sole parameter to the type system because then we would miss out something on the other side. In Chapter 5, we overcome this problem by parametrizing the type system with cartesian product of these structures.

### 1.3.2 Comonadic vs. Monadic

In our discussion on linear type systems, we pointed out that they grade the exponential modality, !, of linear logic to analyze usage. In our discussion on dependency type systems (DCC in particular), we pointed out that they grade the monadic modality, $T$, of Moggi's computational metalanguage to analyze dependency. To unify linearity and dependency analyses, we need to understand how these graded modalities relate to one another. The exponential and monadic modality, while similar in some ways, are also different in other ways.

The exponential modality is comonadic whereas the monadic modality, as the name suggests, is monadic [MacLane, 1971]. Monads and comonads are standard category theoretic constructions that are dual to each other. A monad on a category $\mathbb{C}$ is an endofunctor, $T : \mathbb{C} \to \mathbb{C}$, along with natural transformations, $\eta : \mathbf{Id} \to T$ and $\mu : T \circ T \to T$, that obey standard identity and associativity axioms. A comonad on a category $\mathbb{C}$ is an endofunctor, $D : \mathbb{C} \to \mathbb{C}$, along with natural transformations, $\epsilon : D \to \mathbf{Id}$ and $\delta : D \to D \circ D$, that obey standard identity and associativity axioms. Note here that $\mathbf{Id}$ is the identity functor on $\mathbb{C}$. In the context of programming languages, a monad $T$ is a type constructor equipped with a return function, $\mathbf{return} : A \to T\ A$, and a join function, $\mathbf{join} : T\ T\ A \to T\ A$, that obey the equivalent axioms; a comonad $D$ is a type constructor equipped with an extract function, $\mathbf{extract} : D\ A \to A$, and a fork function, $\mathbf{fork} : D\ A \to D\ D\ A$ that obey the equivalent axioms.

While the exponential and monadic modalities in and of themselves behave like comonads and monads, their graded versions, used in linear and dependency type systems, behave like graded comonads and graded monads respectively. Graded comonads and graded monads [Fujii, 2019] are basically comonads and monads graded over algebraic structures, usually preordered monoids. Chapter 2 presents their formal definitions. In the context of programming languages, a graded monad $T$ over a preordered monoid, $\mathcal{M} = (M, \cdot, 1)$, is a type constructor, graded by $m \in M$, and equipped with a return function, $\mathbf{return} : A \to T_1\ A$, and a graded join function, $\mathbf{join}^{m_1,m_2} : T_{m_1}\ T_{m_2}\ A \to T_{m_1 \cdot m_2}\ A$, for any $m_1, m_2 \in M$, that obey the graded versions of the standard monad axioms. Similarly, a graded comonad $D$ over a preordered monoid, $\mathcal{M} = (M, \cdot, 1)$, is a type constructor, graded by $m \in M$, and equipped with an extract function, $\mathbf{extract} : D_1\ A \to A$, and a graded fork function, $\mathbf{fork}^{m_1,m_2} : D_{m_1 \cdot m_2}\ A \to D_{m_1}\ D_{m_2}\ A$, for any $m_1, m_2 \in M$, that obey the graded versions of the standard comonad axioms.

Now, given a linear type system, parametrized by a preordered semiring $\mathcal{Q} = (Q, +, \cdot, 0, 1, <:)$, its graded exponential modality behaves like a graded comonad over the preordered monoid $(Q, \cdot, 1\ <:)$. Such a type system can derive a graded fork function: $\mathbf{fork}^{q_1,q_2} : !_{q_1 \cdot q_2}\ A \to !_{q_1}\ !_{q_2}\ A$, for any $q_1, q_2 \in Q$. However, it does not derive a graded join function over $(Q, \cdot, 1\ <:)$. Again, given a dependency type system (especially DCC), parametrized by a lattice $\mathcal{L} = (L, \sqcap, \sqcup, \top, \bot, \sqsubseteq)$, its graded monadic modality behaves like a graded monad over the preordered monoid $(L, \sqcup, \bot, \sqsubseteq)$. Such a type system can derive a graded join function: $\mathbf{join}^{\ell_1,\ell_2} : T_{\ell_1}\ T_{\ell_2}\ A \to T_{\ell_1 \sqcup \ell_2}\ A$, for any $\ell_1, \ell_2 \in L$. However, it does not derive a graded fork function over $(L, \sqcup, \bot, \sqsubseteq)$. Note here that the graded fork function is essential for usage analysis and the graded join function is essential

for dependency analysis. But linear and dependency type systems do not derive both these functions.

Our unified calculus, LDC, resolves this problem through a modality that is simultaneously graded comonadic and graded monadic. LDC can derive both graded join and graded fork functions for this modality and that helps unify linearity and dependency analyses. We present the details of this construction in Chapter 5.

## 1.4 Contributions

In this chapter, we have given an overview of the following: the problems we solve, why these problems are important, what makes them challenging and how we have built upon existing work. Next, we summarize our contributions:

- We [Choudhury, 2022a,b] present $\text{GMCC}_\text{e}$, a calculus for dependency analysis in simple type systems. $\text{GMCC}_\text{e}$ addresses some of the limitations of DCC [Abadi et al., 1999], the standard calculus on this subject.
- We [Choudhury et al., 2022a,b] present DDC, a calculus for dependency analysis in pure type systems. DDC can analyze fine-grained notions of irrelevance that are out of reach of existing calculi.
- We [Choudhury et al., 2020, 2021] present GrAD, a calculus for linearity analysis in pure type systems. GrAD addresses some of the shortcomings of QTT [Atkey, 2018], the first-of-its-kind calculus on this subject.
- We [Choudhury, 2022c, Under Review] present LDC, a calculus for combined linearity and dependency analysis in pure type systems. To the best of our knowledge, LDC is the first calculus that combines linearity analysis with a general dependency analysis in pure type systems.

The rest of the dissertation is organized as follows. Chapters 2, 3, 4 and 5 present calculi $\text{GMCC}_\text{e}$, DDC, GrAD and LDC respectively. Chapter 6 concludes the dissertation.

# Chapter 2

# Dependency Analysis in Simple Type Systems

Abadi et al. [1999] presented a general calculus for dependency analyses in simple type systems. Over the years, their calculus has become the standard on this subject. Their calculus, DCC, is a simple extension of Moggi's monadic metalanguage [Moggi, 1991]. The monadic metalanguage is a general calculus for analyzing computational effects. Moggi showed that computational effects in programming languages can be understood in terms of monads from category theory [MacLane, 1971]. At first sight, computational effects seem to be quite different from dependencies. So, it comes as a surprise (pointed out by Abadi et al. [1999] themselves) that with just a simple extension, a calculus for analyzing computational effects can also analyze dependencies.

In this regard, Abadi et al. [1999] point out a common feature underlying monads and security levels: just as there is no way of projecting out of a 'monad world', there is also no way of projecting out of a 'high-security world'. Concretely, just as there is no general function of type $TA \to A$ for a monad $T$ and a type $A$, there is also no non-trivial function from high-security data to low-security variables. This analogy explains the monadic aspect of dependency analysis.

However, the monadic aspect of dependency analysis might be just half of the story. Everyday experience shows us that security constraints can be enforced not only by restricting outflow but also by restricting inflow. For example, in a world with two levels, **Low** and **High**, security can be enforced not only by restricting projection out of **High** level, but also by restricting injection into **Low** level. These ways are dual to one another. The way of restricting projection, employed in DCC, goes via monads. On the other hand, the way of restricting injection goes via comonads. Similar to a monad $T$ that restricts by disallowing a general function of type $TA \to A$, a comonad $D$ restricts by disallowing a general function of type $A \to DA$. While the monadic aspect of dependency analysis has received considerable attention in literature, the comonadic aspect has received less so.

In this chapter, we show that just like the monadic aspect, the comonadic aspect of dependency analysis also has much to offer. Going further, we show that the monadic and the comonadic aspects play nicely with one another. We design a language that integrates these two aspects into a single system. This integration helps us unify DCC, a monadic dependency calculus and $\lambda^\circ$ [Davies, 2017], a comonadic dependency calculus (that is known to be outside the reach of DCC [Abadi et al., 1999]). It also leads to a novel technique for proving correctness of dependency analysis. Above all, it shines light on some of the nuances of dependency analysis.

In short, we make the following contributions in this chapter:

- We present a Graded Monadic Comonadic Calculus, GMCC, and its extension, $\text{GMCC}_\text{e}$, and provide meaning-preserving translations from both DCC and $\lambda^\circ$ to $\text{GMCC}_\text{e}$.
- We show that the protection judgment of DCC, when appropriately modified, enables comonadic reasoning in the language. Further, we show that under certain restrictions, DCC, with this modification, is equivalent to GMCC. This equivalence helps explain the nonstandard bind-rule of DCC in terms of standard categorical concepts.
- GMCC and $\text{GMCC}_\text{e}$ are general calculi that are sound with respect to a class of categorical models. These categorical models motivate a novel technique for proving correctness of dependency analyses. We use this technique to provide simple proofs of correctness for both DCC and $\lambda^\circ$.

In the next section, we review the basics of dependency analysis and its application in information flow control and staged execution of programs. This section is meant to provide some background to readers who are not very familiar with dependency analysis.

## 2.1 Dependency Analysis in Action

Consider a database `Db` containing demographic information of a city. For the sake of simplicity, let's say the database is represented as a list of tuples with elements of the list corresponding to residents of the city and elements of the tuples corresponding to their demographic information. Further, let's assume that each tuple has only 4 elements for recording name, age, ethnicity and monthly income of a resident (in that order). According to the policies of the city council, name, age and ethnicities of the residents are non-sensitive information that may be shared without any constraint, whereas monthly income of the residents is sensitive information that may be shared only with people who are allowed to handle such information.

Now, consider the following queries:

1. What fraction of the elderly city residents (age $\geq$ 65 years) are ethnically Caucasian?
2. What is the average monthly income of ethnically Asian residents of the city?

With the above queries in mind, a programmer seeks the outputs of the following programs, written in `Haskell`-like syntax:

1.
```
lstEld = filter (\ x -> second x >= 65) Db
lstEldC = filter (\ x -> third x == "Caucasian") lstEld
fracEldC = (length lstEldC) / (length lstEld)
print fracEldC
```

2.
```
lstAsn = filter (\ x -> third x == "Asian") Db
totIncA = foldr (\ x y -> fourth x + y) 0 lstAsn
avgIncA = totIncA / (length lstAsn)
print avgIncA
```

Note that functions `second`, `third` and `fourth` access the second, third and fourth elements of a tuple respectively. The question we need to address now is whether the programmer may be allowed to see the outputs of the above programs. Let's suppose this programmer does not have the permission to handle sensitive information. Then, the output of the second program should not be shared with this programmer because it reveals (at least partially) monthly income data. For example, if there's just a single ethnically Asian resident in the city, then the output of the second program gives away the monthly income of that resident. The output of the first program may, however, be shared freely with this programmer because it does not reveal any sensitive information.

Next, how do we reach this conclusion about sharing output by *just* analysing the respective programs? In other words, given a program, how do we decide whether or not its output reveals any sensitive information? To answer this question, we need to perform a dependency analysis called information flow analysis: If the output of a program *depends upon* any information deemed sensitive, then the output too needs to be treated as sensitive. Conversely, if the output *does not depend upon* any sensitive information, then the output too is not sensitive. In the second program above, the output, `avgIncA`, is sensitive because it *depends upon* `totIncA`, which in turn *depends upon* `fourth x`, a sensitive piece of information. On the other hand, in the first program, all the data used to compute the output are non-sensitive, rendering the output itself non-sensitive.

To analyze dependency of output upon sensitive information, information flow calculi typically incorporate sensitivity of information in the types themselves. For example, the type of `Db` would be `[(T L String, T L Int, T L String, T H Int)]`, where `T l String` and `T l Int` are the types of `l`-security strings and integers respectively, with `l` being `H` or `L`, corresponding to high and low security respectively. In information flow calculi, functions, too, have types

that take sensitivity of information into account. For example, the type of '+' would be `T l Int -> T l Int -> T l Int`. Now, if we write the above programs in such a calculus, we would see that `fracEldC` has type `T L Double` whereas `avgIncA` has type `T H Double`. Any user can access terms of type `T L Double` but terms of type `T H Double` are only accessible to users with high-security clearance. Within the calculus, the noninterference property enforces this restriction. In this way, information flow calculi ensure secure flow of information.

Our next example takes up another form of dependency analysis: binding-time analysis. Binding-time analysis helps in staged execution of programs. Staged execution comes in handy when programs have inputs that are statically known in addition to inputs that are known only at run-time. Computations that depend only upon static inputs may be carried out statically, thereby producing residual programs that can be executed faster at run-time. To find out which computations depend only upon static inputs, we perform a dependency analysis, called binding-time analysis.

Next, we shall use the same database example to show binding-time analysis in action. However, instead of sensitivity of information, here we focus on availability of information. From the given demographic parameters, name and ethnicity remain constant over time whereas monthly income varies. To account for this fact, the city council mandates that every resident update their monthly income on the last day of each month. Now, to get an accurate answer to the second query, one needs to run the second program on these days. But if the database contains millions of entries, running this program may burden the computing system on these days. However, observe that some or perhaps most of the work done by this program need not wait for the update in monthly incomes. For example, `lstAsn` can be computed statically beforehand because this computation depends only upon static information. Binding-time analysis identifies such computations, thereby enabling faster execution of programs.

Binding-time calculi typically incorporate information about binding-time in the types themselves. For example, the type of `Db` would be `[(T Sta String, T Sta Int, T Sta String, T Dyn Int)]`, where `T l String` and `T l Int` are respectively the types of strings and integers available at time `l`, with `l` being `Dyn` or `Sta`, corresponding to dynamic and static availability respectively. In binding-time calculi, functions, too, have types that take binding-time information into account. For example, the type of '+' would be `T l Int -> T l Int -> T l Int`. Now, if we write the second program in such a calculus, we would see that the computation of `lstAsn` depends upon static data only (To compute `totIncA`, however, we would need dynamic data, `fourth x`). So we could compute `lstAsn` statically and thereafter run *only the residual program* on the last day of every month, thereby reducing the burden on the computing system on such days. To give an estimate, if ethnically Asian residents constitute 5% of the total population (say), then, compared to the original program, the residual program may run 20x faster.

With this background on dependency analysis, we shall now work towards building our dependency calculi. We shall present two key dependency calculi in this chapter: GMCC and its extension, $\text{GMCC}_e$. The calculus GMCC is built up from a graded monadic calculus, GMC,

and a graded comonadic calculus, GCC. In the next section, we look at the Graded Monadic Calculus (GMC).

## 2.2 Graded Monadic Calculus

Moggi [Moggi, 1991] showed that computational effects can be understood in terms of monads. On the other hand, Gifford and Lucassen [1986] showed that side effects can be tracked using effect classes. Wadler and Thiemann [2003] later adapted effect classes to monads but left open the question of a general theory of effects and monads. Eventually, Katsumata [2014] presented such a general theory in terms of graded monads. In this section, we adapt Katsumata's Explicit Subeffecting Calculus to present a simply-typed Graded Monadic Calculus (GMC).

GMC is an extension of the simply-typed $\lambda$-calculus with a grade-annotated monadic type constructor, $T_m$. The grades, $m$, are drawn from an arbitrary preordered monoid $\mathcal{M} = (M, \_\cdot\_, 1, \leq)$. Recall that a preordered monoid $\mathcal{M}$ is a monoid $(M, \_\cdot\_, 1)$ along with a preorder $\leq$ such that the order respects the binary operator, meaning if $m_1 \leq m_1'$ and $m_2 \leq m_2'$, then $m_1 \cdot m_2 \leq m_1' \cdot m_2'$. Note that whenever we need to be precise, we use GMC($\mathcal{M}$) to refer to GMC parametrized by $\mathcal{M}$. We follow the same convention for other calculi parametrized by algebraic structures.
Next, we present the calculus formally.

### 2.2.1 Grammar and Type System

The grammar of the calculus appears in Figure 2.1. In addition to the types and terms of standard $\lambda$-calculus, we have a graded monadic type, $T_m\ A$, and terms related to it. The typing rules of the calculus appear in Figure 2.2. We omit the typing rules of standard $\lambda$-calculus and consider only the ones related to the graded monadic type.

$$
\begin{aligned}
\text{types}, A, B &::= \mathbf{Unit} \mid \mathbf{Void} \mid A \to B \mid A \times B \mid A + B \mid T_m\ A \\
\text{terms}, a, b, f, g &::= x \mid \lambda x : A.b \mid b\ a \\
&\quad \mid (a_1, a_2) \mid \mathbf{proj}_1\ a \mid \mathbf{proj}_2\ a \mid \mathbf{unit} \\
&\quad \mid \mathbf{inj}_1\ a_1 \mid \mathbf{inj}_2\ a_2 \mid \mathbf{case}\ a\ \mathbf{of}\ b_1\ ;\ b_2 \mid \mathbf{abort}\ a \\
&\quad \mid \mathbf{ret}\ a \mid \mathbf{lift}^m f \mid \mathbf{join}^{m_1,m_2}\ a \mid \mathbf{up}^{m_1,m_2}\ a \\
\text{contexts}, \Gamma &::= \varnothing \mid \Gamma, x : A
\end{aligned}
$$

FIGURE 2.1: Grammar of GMC

The rules M-RETURN, M-FMAP, and M-JOIN are generalizations of the corresponding rules for the ungraded monadic type. Note that the rule M-JOIN 'joins' the grades using the binary operator of the monoid. The rule M-UP relaxes the grade on the monadic type. If $\mathcal{M}$ is the

$\boxed{\Gamma \vdash a : A}$ *(Typing rules)*

$$\text{M-Return}\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{ret}\ a : T_1\ A} \qquad \text{M-Fmap}\ \frac{\Gamma \vdash f : A \to B}{\Gamma \vdash \mathbf{lift}^m f : T_m\ A \to T_m\ B} \qquad \text{M-Join}\ \frac{\Gamma \vdash a : T_{m_1}\ T_{m_2}\ A}{\Gamma \vdash \mathbf{join}^{m_1,m_2}\ a : T_{m_1 \cdot m_2}\ A}$$

$$\text{M-Up}\ \frac{\Gamma \vdash a : T_{m_1}\ A \qquad m_1 \leq m_2}{\Gamma \vdash \mathbf{up}^{m_1,m_2}\ a : T_{m_2}\ A}$$

FIGURE 2.2: Typing rules of GMC (excerpt)

$$\mathbf{lift}^m(\lambda x.x) \equiv \lambda x.x \tag{2.1}$$
$$\mathbf{lift}^m(\lambda x.g\ (f\ x)) \equiv \lambda x.(\mathbf{lift}^m\ g)\ ((\mathbf{lift}^m f)\ x) \tag{2.2}$$
$$\mathbf{up}^{m_1,m_1}\ a \equiv a \tag{2.3}$$
$$\mathbf{up}^{m_2,m_3}(\mathbf{up}^{m_1,m_2}\ a) \equiv \mathbf{up}^{m_1,m_3}\ a \tag{2.4}$$
$$(\mathbf{up}^{m_1,m_1'}\ a)\ {}^{m_1'}{\gg\!=}^{m_2}\ f \equiv \mathbf{up}^{m_1 \cdot m_2, m_1' \cdot m_2}(a\ {}^{m_1}{\gg\!=}^{m_2}\ f) \tag{2.5}$$
$$a\ {}^{m_1}{\gg\!=}^{m_2'}\ (\lambda x.\mathbf{up}^{m_2,m_2'}\ b) \equiv \mathbf{up}^{m_1 \cdot m_2, m_1 \cdot m_2'}(a\ {}^{m_1}{\gg\!=}^{m_2}\ \lambda x.b) \tag{2.6}$$
$$(\mathbf{ret}\ a)\ {}^{1}{\gg\!=}^{m}\ f \equiv f\ a \tag{2.7}$$
$$a\ {}^{m_1}{\gg\!=}^{1}\ (\lambda x.\mathbf{ret}\ x) \equiv a \tag{2.8}$$
$$(a\ {}^{m_1}{\gg\!=}^{m_2}\ f)\ {}^{m_1 \cdot m_2}{\gg\!=}^{m_3}\ g \equiv a\ {}^{m_1}{\gg\!=}^{m_2 \cdot m_3}\ (\lambda x.(f\ x\ {}^{m_2}{\gg\!=}^{m_3}\ g)) \tag{2.9}$$

FIGURE 2.3: Equality rules of GMC (excerpt)

trivial preordered monoid, then the above rules degenerate to the standard typing rules for monads.
Next, we look at the equational theory of the calculus.

### 2.2.2 Equational Theory

Equality over terms of the graded monadic calculus is a congruent equivalence relation generated by the standard $\beta\eta$-equality rules over $\lambda$-terms and the additional rules that appear in Figure 2.3. For presenting the rules, we use the shorthand notation: $a\ {}^{m_1}{\gg\!=}^{m_2}\ f \triangleq \mathbf{join}^{m_1,m_2}((\mathbf{lift}^{m_1} f)\ a)$ where $a : T_{m_1}\ A$ and $f : A \to T_{m_2}\ B$. Note that $\_\,{}^{m_1}{\gg\!=}^{m_2}\,\_$ is a graded **bind**-operator.

The first two rules correspond to preservation of identity function and composition of functions by **lift**. The next two rules correspond to reflexivity and transitivity of the order relation. The two rules after that correspond to commutativity of **bind** with **up**. The two subsequent rules correspond to **ret** being the left and the right identity of **bind**. The last rule corresponds to associativity of **bind**.

We now want to interpret this calculus in a suitable category. The types of standard $\lambda$-calculus can be interpreted in any bicartesian closed category. To interpret the graded monadic type, we need a graded monad. Fujii [2019] provides a nice account on graded monads and graded comonads. However, for the sake of self-containment, we shall briefly review the theory behind graded monads in the following section.

### 2.2.3 Graded Monads

A graded monad is a certain kind of lax monoidal functor [MacLane, 1971]. A lax monoidal functor from a monoidal category $(M, \otimes_M, 1_M)$ to a monoidal category $(N, \otimes_N, 1_N)$ is a 3-tuple $(F, F_2, F_0)$ where,

- $F$ is a functor from $M$ to $N$
- For $X, Y \in \mathrm{Obj}(M)$, morphisms $F_2(X,Y) : F(X) \otimes_N F(Y) \to F(X \otimes_M Y)$ are natural in $X$ and $Y$
- $F_0 : 1_N \to F(1_M)$

such that the diagrams in Figure 2.4 commute.

$$\begin{array}{ccccc} F(1_M) \otimes_N F(X) & \xleftarrow{F_0 \otimes_N \mathrm{id}} & F(X) & \xrightarrow{\mathrm{id} \otimes_N F_0} & F(X) \otimes_N F(1_M) \\ & {\scriptstyle F_2(1_M, X)} \searrow & \downarrow {\scriptstyle \mathrm{id}} & \swarrow {\scriptstyle F_2(X, 1_M)} & \\ & & F(X) & & \end{array}$$

$$\begin{array}{ccc} F(X) \otimes_N F(Y) \otimes_N F(Z) & \xrightarrow{\mathrm{id} \otimes_N F_2(Y,Z)} & F(X) \otimes_N F(Y \otimes_M Z) \\ {\scriptstyle F_2(X,Y) \otimes_N \mathrm{id}} \downarrow & & \downarrow {\scriptstyle F_2(X, Y \otimes_M Z)} \\ F(X \otimes_M Y) \otimes_N F(Z) & \xrightarrow{F_2(X \otimes_M Y, Z)} & F(X \otimes_M Y \otimes_M Z) \end{array}$$

FIGURE 2.4: Commutative diagrams for lax monoidal functor

Note that here we assume $M$ and $N$ to be strict monoidal categories, i.e. $1_M \otimes_M X = X = X \otimes_M 1_M$ and $(X \otimes_M Y) \otimes_M Z = X \otimes_M (Y \otimes_M Z)$ for any $X, Y, Z \in \mathrm{Obj}(M)$, and similarly for $N$.

An example of a strict monoidal category is a preordered monoid $\mathcal{M}$. Any preorder $(M, \leq)$ may be seen as a category, $\mathbf{C}((M, \leq))$, that has $M$ as its set of objects and a unique morphism from $m_1$ to $m_2$ if and only if $m_1 \leq m_2$. Identity morphisms and composition of morphisms are given by reflexivity and transitivity of the order relation. Now, a preordered monoid $\mathcal{M} = (M, \_\cdot\_, 1, \leq)$ may be seen as a strict monoidal category: $\mathbf{C}(\mathcal{M}) = (\mathbf{C}((M, \leq)), \_\cdot\_, 1)$.

Another example of a strict monoidal category is the category of endofunctors. Given any category $\mathbb{C}$, the endofunctors of $\mathbb{C}$ form a strict monoidal category, $\mathbf{End}_{\mathbb{C}}$, with the tensor

product given by composition, $\_ \circ \_$, of functors and the identity object given by the identity functor, **Id**.

We use these two monoidal categories to define a graded monad. An $\mathcal{M}$-graded monad over $\mathbb{C}$ is a lax monoidal functor from $\mathbf{C}(\mathcal{M})$ to $\mathbf{End}_{\mathbb{C}}$. We wish to use an $\mathcal{M}$-graded monad to interpret GMC($\mathcal{M}$). However, such a monad doesn't stand up to the task (Try interpreting rule M-FMAP!). This shortcoming should not come as a surprise because we know that monads, in and of themselves, cannot model the monadic type constructor of Moggi's computational metalanguage [Moggi, 1991]. For that, they need to be accompanied with tensorial strengths. Here too, we need to add tensorial strengths to graded monads to interpret the graded monadic type constructor. One could define tensorial strength separately after having defined a graded monad first. However, in lieu, one can also just define a strong graded monad in one go using the category of strong endofunctors and strong natural transformations.

An endofunctor $F$ on a monoidal category $(M, \otimes, 1, \alpha, \lambda, \rho)$ is said to be strong [Eilenberg and Kelly, 1966, Kock, 1970, 1972] if there exists morphisms $t_{X,Y} : X \otimes F(Y) \to F(X \otimes Y)$, natural in $X$ and $Y$, for $X, Y \in \mathrm{Obj}(M)$, such that the diagrams in Figure 2.5 commute.

$$\begin{array}{ccccc}
(X \otimes Y) \otimes FZ & & \xrightarrow{\quad t_{X\otimes Y,Z} \quad} & & F((X \otimes Y) \otimes Z) \\
{\scriptstyle \alpha^{-1}_{X,Y,FZ}}\downarrow & & & & \downarrow{\scriptstyle F\alpha^{-1}_{X,Y,Z}} \\
X \otimes (Y \otimes FZ) & \xrightarrow{\mathrm{id}\otimes t_{Y,Z}} & X \otimes F(Y \otimes Z) & \xrightarrow{t_{X,Y\otimes Z}} & F(X \otimes (Y \otimes Z))
\end{array}$$

$$\begin{array}{ccc}
1 \otimes FX & \xrightarrow{t_{1,X}} & F(1 \otimes X) \\
& {\scriptstyle \lambda_{FX}} \searrow & \downarrow{\scriptstyle F\lambda_X} \\
& & FX
\end{array}$$

FIGURE 2.5: Commutative diagrams for strong endofunctor

Given strong endofunctors $(F, t^F)$ and $(G, t^G)$, a natural transformation $\alpha : F \to G$ is said to be strong, if for any $X, Y \in \mathrm{Obj}(M)$, the diagram in Figure 2.6 commutes. Strong endofunctors can be defined for arbitrary monoidal categories; however, we just need the ones over cartesian monoidal categories. Given any cartesian category $\mathbb{C}$, let $\mathbf{End}^s_{\mathbb{C}}$ denote the category with objects: strong endofunctors over $(\mathbb{C}, \times, \top)$ (where $\top$ is the terminal object) and morphisms: strong natural transformations between them. Like $\mathbf{End}_{\mathbb{C}}$, category $\mathbf{End}^s_{\mathbb{C}}$ too is strict monoidal with the monoidal product and the identity object defined in the same way.

$$\begin{array}{ccc}
X \otimes FY & \xrightarrow{\mathrm{id}\otimes\alpha_Y} & X \otimes GY \\
{\scriptstyle t^F_{X,Y}}\downarrow & & \downarrow{\scriptstyle t^G_{X,Y}} \\
F(X \otimes Y) & \xrightarrow{\alpha_{X\otimes Y}} & G(X \otimes Y)
\end{array}$$

FIGURE 2.6: Commutative diagram for strong natural transformation

Finally, we have the definition of a strong graded monad. Given a preordered monoid $\mathcal{M}$ and a cartesian category $\mathbb{C}$, a strong $\mathcal{M}$-graded monad over $\mathbb{C}$ is a lax monoidal functor from $\mathbf{C}(\mathcal{M})$ to $\mathbf{End}^s_{\mathbb{C}}$. Using strong graded monads, we can now provide a categorical model for the graded monadic calculus.

### 2.2.4 Categorical Model

Let $\mathbb{C}$ be any bicartesian closed category. Let $(\mathbf{T}, \mu, \eta)$ be a strong $\mathcal{M}$-graded monad over $\mathbb{C}$. Then, the interpretation, $[\![\_]\!]$, of types and terms of GMC($\mathcal{M}$) is as follows: The types and terms of standard $\lambda$-calculus are interpreted in the usual way. The graded monadic type and terms related to it are interpreted in Figure 2.7.

$$
\begin{aligned}
[\![T_m\ A]\!] &= \mathbf{T}_m[\![A]\!] & [\![\mathbf{ret}\ a]\!] &= \eta \circ [\![a]\!] \\
[\![\mathbf{lift}^m f]\!] &= \Lambda(\mathbf{T}_m(\Lambda^{-1}[\![f]\!]) \circ t^{\mathbf{T}_m}) & [\![\mathbf{join}^{m_1,m_2}\ a]\!] &= \mu^{m_1,m_2} \circ [\![a]\!] \\
[\![\mathbf{up}^{m_1,m_2}\ a]\!] &= \mathbf{T}^{m_1 \leq m_2} \circ [\![a]\!]
\end{aligned}
$$

FIGURE 2.7: Interpretation of GMC (excerpt)

There are a few things to note here:

- $\mathbf{T}(m)$, written as $\mathbf{T}_m$, is a functor
- $\mathbf{T}(m_1 \leq m_2)$, written as $\mathbf{T}^{m_1 \leq m_2}$, is a natural transformation
- $\eta$ is a natural transformation from $\mathbf{Id}$ to $\mathbf{T}_1$
- $\mu^{m_1,m_2}$ are morphisms from $\mathbf{T}_{m_1} \circ \mathbf{T}_{m_2}$ to $\mathbf{T}_{m_1 \cdot m_2}$, and are natural in both $m_1$ and $m_2$
- $t^{\mathbf{T}_m}$ denotes the strength of $\mathbf{T}_m$
- $\Lambda$ and $\Lambda^{-1}$ denote currying and uncurrying respectively

Let us now see why this interpretation satisfies the equational theory of the calculus. Equations (2.1) and (2.2) follow because $\mathbf{T}_m$ is a functor, for any $m \in M$. Equations (2.3) and (2.4) follow because $\mathbf{T}$ is a functor and as such, preserves identity morphisms and composition of morphisms. Equations (2.5) and (2.6) follow because $\mu$ is natural in its first component and its second component respectively. Equations (2.7) and (2.8) follow respectively from the left and right unit laws for graded monad, laws that correspond to the commutative triangles in Figure 2.4. Equation (2.9) follows from the associative law for graded monad, the law that corresponds to the commutative square in Figure 2.4. Note that soundness of the equations in the model also depends upon the axioms about strength, shown in Figures 2.5 and 2.6.

Thus, a bicartesian closed category, $\mathbb{C}$, along with a strong $\mathcal{M}$-graded monad over $\mathbb{C}$, gives us a sound model of GMC($\mathcal{M}$).

$\boxed{\ell \sqsubseteq A}$ *(DCC Protect)*

$$\textsc{Prot-Prod}\ \frac{\ell \sqsubseteq A \qquad \ell \sqsubseteq B}{\ell \sqsubseteq A \times B} \qquad \textsc{Prot-Fun}\ \frac{\ell \sqsubseteq B}{\ell \sqsubseteq A \to B} \qquad \textsc{Prot-Monad}\ \frac{\ell_1 \sqsubseteq \ell_2}{\ell_1 \sqsubseteq \mathcal{T}_{\ell_2}\ A} \qquad \textsc{Prot-Already}\ \frac{\ell \sqsubseteq A}{\ell \sqsubseteq \mathcal{T}_{\ell'}\ A}$$

FIGURE 2.8: Protection rules for DCC

**Theorem 2.1** If $\Gamma \vdash a : A$ in GMC, then $[\![a]\!] \in \mathrm{Hom}_{\mathbb{C}}([\![\Gamma]\!], [\![A]\!])$. Further, if $\Gamma \vdash a_1 : A$ and $\Gamma \vdash a_2 : A$ such that $a_1 \equiv a_2$ in GMC, then $[\![a_1]\!] = [\![a_2]\!] \in \mathrm{Hom}_{\mathbb{C}}([\![\Gamma]\!], [\![A]\!])$.

## 2.3 DCC and GMC

In this section, we look at the relation between DCC and GMC. Since DCC lies at the heart of this chapter, we next review the calculus briefly. For simplicity, we first focus on the terminating fragment of the calculus and consider non-termination later in this chapter.

### 2.3.1 Dependency Core Calculus

The Dependency Core Calculus is simply-typed $\lambda$-calculus, extended with multiple type constructors, $\mathcal{T}_\ell$, which help analyze dependencies. The indices, $\ell$, are elements of an abstract lattice $\mathcal{L} = (L, \sqcup, \sqcap, \bot, \top)$. The lattice structure for the calculus is motivated by the lattice model of secure information flow [Denning, 1976]. The elements of a lattice model may be thought of as dependency levels, with $\ell_1 \sqsubseteq \ell_2$ meaning $\ell_2$ may depend upon $\ell_1$ and $\neg(\ell_1 \sqsubseteq \ell_2)$ meaning $\ell_2$ should not depend upon $\ell_1$. (Here, $\sqsubseteq$ is the implied order of the lattice.) For example, low and high security levels may be modeled using a two-point lattice $\mathcal{L}_2$: **Low** $\sqsubset$ **High**.

DCC uses an auxiliary protection judgment to analyze dependency. The protection judgment, written $\ell \sqsubseteq A$, and presented in Figure 2.8, can be read as: the terms of type $A$ may depend upon level $\ell$. With this reading of the protection judgment, the calculus may be said to be correct when it satisfies the following condition: if $\ell \sqsubseteq A$ and $\neg(\ell \sqsubseteq \ell')$, then the terms of type $A$ 'should not be visible' at $\ell'$. We formalize this condition in Section 2.8.2. Let us now look at the type system and equational theory of DCC.

### 2.3.2 Type System and Equational Theory of DCC

The typing rules of DCC consist of the ones for standard $\lambda$-calculus along with the introduction and elimination rules for $\mathcal{T}_\ell$, shown below.

$\boxed{\Gamma \vdash a : A}$ *(DCC Typing (Excerpt))*

$$\text{DCC-Eta}\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{eta}^{\ell}\, a : \mathcal{T}_{\ell}\, A} \qquad \text{DCC-Bind}\ \frac{\Gamma \vdash a : \mathcal{T}_{\ell}\, A \qquad \Gamma, x : A \vdash b : B \qquad \ell \sqsubseteq B}{\Gamma \vdash \mathbf{bind}^{\ell}\, x = a\, \mathbf{in}\, b : B}$$

The protection judgment in rule DCC-Bind ensures that $a$ is visible to $B$ only if $B$ has the necessary permission. Note that rule DCC-Bind, unlike a standard monadic bind rule, does not wrap the return type, $B$, with the constructor $\mathcal{T}_{\ell}$. This difference is significant and we shall see its implications as we go along.

Now we consider the equational theory of DCC. Abadi et al. [1999] do not explicitly provide an equational theory for DCC. However, they provide an operational semantics for DCC. We describe the equational theory corresponding to the operational semantics they provide. The terms of DCC can be seen as $\lambda$-terms annotated with security labels. If we erase the annotations, we are left with plain $\lambda$-terms. Plain $\lambda$-terms already have an equational theory: the one generated by the standard $\beta\eta$-rules. Using this theory, we define the equational theory of DCC as follows: two DCC terms are equal, if and only if, after erasure, they are equal as $\lambda$-terms, i.e. $a_1 \simeq a_2 \triangleq \lfloor a_1 \rfloor \equiv \lfloor a_2 \rfloor$, where $\lfloor a \rfloor$ is the plain $\lambda$-term corresponding to the DCC-term $a$.

Now we are in a position to explore the relation between DCC and GMC. There are two questions that we would like to address.

- Is DCC a graded monadic calculus? In other words, with appropriate restrictions, can we translate GMC to DCC while preserving meaning?
- Is DCC just a graded monadic calculus? In other words, with appropriate restrictions, can we translate DCC to GMC while preserving meaning?

We shall see that the answer to the first question is yes, while the answer to the second one is no.

### 2.3.3 Is DCC a Graded Monadic Calculus?

Both DCC and GMC are calculi parametrized by algebraic structures. To compare the calculi, we first need to relate the parametrizing structures. DCC is parametrized by an arbitrary lattice $\mathcal{L}$ whereas GMC is parametrized by an arbitrary preordered monoid $\mathcal{M}$. A preordered monoid is a more general structure because any bounded semilattice may be seen as a preordered monoid. For example, a bounded join-semilattice is a preordered monoid with multiplication, unit and the preorder given by join, $\bot$ and the semilattice order respectively. A point to note here is that in the original formulation of Denning [1976], the semantics of secure information flow just

$$
\begin{aligned}
\overline{T_\ell\ A} &= \mathcal{T}_\ell\ \overline{A} \\
\overline{\mathbf{ret}\ a} &= \mathbf{eta}^{\perp}\overline{a} \\
\overline{\mathbf{lift}^{\ell} f} &= \lambda x : \mathcal{T}_\ell\ \overline{A}.\mathbf{bind}^{\ell}\ y = x\ \mathbf{in}\ \mathbf{eta}^{\ell}(\overline{f}\ y)\ \text{[Here, } f : A \to B\text{]} \\
\overline{\mathbf{join}^{\ell_1,\ell_2}\, a} &= \mathbf{bind}^{\ell_1}\ x = \overline{a}\ \mathbf{in}\ \mathbf{bind}^{\ell_2}\ y = x\ \mathbf{in}\ \mathbf{eta}^{\ell_1 \sqcup \ell_2} y \\
\overline{\mathbf{up}^{\ell_1,\ell_2}\, a} &= \mathbf{bind}^{\ell_1}\ x = \overline{a}\ \mathbf{in}\ \mathbf{eta}^{\ell_2} x
\end{aligned}
$$

FIGURE 2.9: Translation function from GMC to DCC (excerpt)

constrains the model to be a bounded join-semilattice. However, under the practical assumption of finiteness, such a model collapses to a lattice. Here, we shall compare DCC and GMC over the class of bounded join-semilattices.

Let $\mathcal{L} = (L, \sqcup, \perp)$ be a bounded join-semilattice. Then, the translation, $\overline{\ }$, from GMC to DCC, is given in Figure 2.9. This translation preserves typing and meaning.

**Theorem 2.2** If $\Gamma \vdash a : A$ in GMC($\mathcal{L}$), then $\overline{\Gamma} \vdash \overline{a} : \overline{A}$ in DCC($\mathcal{L}$). Further, if $\Gamma \vdash a_1 : A$ and $\Gamma \vdash a_2 : A$ such that $a_1 \equiv a_2$ in GMC($\mathcal{L}$), then $\overline{a_1} \simeq \overline{a_2}$ in DCC($\mathcal{L}$).

### 2.3.4 Is DCC Just a Graded Monadic Calculus?

Now that GMC can be translated into DCC, can we go the other way around? Let's explore this question. To translate DCC to GMC, we would need to translate the **bind** construct. We may attempt a translation for **bind** of DCC using **bind** of GMC. However, note that the signature of **bind** in GMC is: $T_{\ell_1}\ A \to (A \to T_{\ell_2}\ B) \to T_{\ell_1 \sqcup \ell_2}\ B$ whereas that of **bind** in DCC is: $\mathcal{T}_{\ell_1}\ A \to (A \to B) \to \{\ell_1 \sqsubseteq B\} \to B$. For a successful translation, one needs to show that, if $\ell_1 \sqsubseteq B$, then there exists a function $j$ of type $(T_{\ell_1}\ \underline{B} \to \underline{B})$. In case such a function exists, for $a : \mathcal{T}_{\ell_1}\ A$ and $f : A \to B$, one can get $(j\ ((\mathbf{lift}^{\ell_1} \underline{f})\ \underline{a})) : \underline{B}$. (Here, _ denotes a possible translation of DCC to GMC.)

We attempt to define $j : T_{\ell_1}\ \underline{B} \to \underline{B}$ by structural recursion on the judgment $\ell_1 \sqsubseteq B$. The interesting cases are rules PROT-MONAD and PROT-ALREADY.

- Rule PROT-MONAD. Here, we have $\ell \sqsubseteq \mathcal{T}_{\ell'} B$ where $\ell \sqsubseteq \ell'$. Need to define $j : T_\ell T_{\ell'} \underline{B} \to T_{\ell'} \underline{B}$. But, $x : T_\ell\ T_{\ell'}\ \underline{B} \vdash \mathbf{join}^{\ell,\ell'} x : T_{\ell'}\ \underline{B}$ because $\ell \sqcup \ell' = \ell'$.

- Rule PROT-ALREADY. Here, we have $\ell \sqsubseteq \mathcal{T}_{\ell'}\ B$ where $\ell \sqsubseteq B$. Need to define $j : T_\ell\ T_{\ell'}\ \underline{B} \to T_{\ell'}\ \underline{B}$. Since $\ell \sqsubseteq B$, the hypothesis gives us a function $j_0 : T_\ell\ \underline{B} \to \underline{B}$. But, now we are stuck! Lifting this function can only give us: $\mathbf{lift}^{\ell'} j_0 : T_{\ell'}\ T_\ell\ \underline{B} \to T_{\ell'}\ \underline{B}$, not exactly what we need.

  Here, we could, for instance, add a non-standard flip-rule to GMC like: "from $\Gamma \vdash a : T_{\ell_1}\ T_{\ell_2}\ A$, derive $\Gamma \vdash \mathbf{flip}^{\ell_1,\ell_2} a : T_{\ell_2}\ T_{\ell_1}\ A$" and thereafter translate DCC into it. But such an exercise would defeat our whole purpose because then, GMC would no longer be a

graded monadic calculus. Note that Algehed [2018] includes such a rule in his language SDCC, which is shown to be equivalent to (the terminating fragment of) DCC.

So we see that DCC is not just a graded monadic calculus. The rule Prot-Already makes it something more than that. This rule enables one to flip the modal type constructors. In DCC, from $\Gamma \vdash a : \mathcal{T}_{\ell_1}\ \mathcal{T}_{\ell_2}\ A$, one can derive $\Gamma \vdash \mathbf{bind}^{\ell_1}\ x = a\ \mathbf{in}\ \mathbf{bind}^{\ell_2}\ y = x\ \mathbf{in}\ \mathbf{eta}^{\ell_2}\mathbf{eta}^{\ell_1}\ y : \mathcal{T}_{\ell_2}\ \mathcal{T}_{\ell_1}\ A$, using rule Prot-Already. Such a derivation is not possible in a general monadic calculus.

However, if the calculus is also comonadic in addition to being monadic, such a derivation is possible. From $\mathcal{T}_{\ell_1}\ \mathcal{T}_{\ell_2}\ A$, using monadic join, we can get $\mathcal{T}_{\ell_1 \sqcup \ell_2}\ A$, which is same as $\mathcal{T}_{\ell_2 \sqcup \ell_1}\ A$, from which we can get $\mathcal{T}_{\ell_2}\ \mathcal{T}_{\ell_1}\ A$, using comonadic fork. So it seems that DCC has some comonadic flavor to it. But is DCC a graded comonadic calculus? In order to address this question, we first need to build the theory of a graded comonadic calculus.

## 2.4 Graded Comonadic Calculus

Soon after Moggi [1991] showed that computational effects can be understood in terms of monads, Brookes and Geva [1992] showed that intensional behaviour of programs, for example, the number of steps necessary for reduction, can be understood in terms of comonads. While monads can model how programs affect the environment, comonads can model how the environment affects programs. Comonads, with necessary extra structure, have been used by Uustalu and Vene [2008] and Petricek et al. [2013], among others, to model various notions of environment-dependent computation like resource usage in programs, computation on streams, etc. Several calculi [Brunel et al., 2014, Ghica and Smith, 2014, Petricek et al., 2014], some of which we have already come across in the last chapter, have been developed to provide a general account of such environment-dependent computation. These calculi are parametrized by semiring-like structures, and are modeled using semiring-graded comonads, possibly including additional structure. In this section, we forgo extra structures and present a graded comonadic calculus that is the dual of the graded monadic calculus presented in Section 2.2.

The Graded Comonadic Calculus (GCC), similar to GMC, is parametrized by a preordered monoid $\mathcal{M}$. In addition to the types and terms of standard $\lambda$-calculus, GCC has a graded comonadic type, $D_m\ A$, and terms related to it, shown below.

$$\begin{aligned} \text{types}, A, B &::= \ldots \mid D_m\ A \\ \text{terms}, a, b, f, g &::= \ldots \mid \mathbf{extr}\ a \mid \mathbf{lift}^m f \mid \mathbf{fork}^{m_1,m_2}\ a \mid \mathbf{up}^{m_1,m_2}\ a \end{aligned}$$

Now we look at the typing rules and equational theory of the calculus.

### 2.4.1 Type System and Equational Theory

The typing rules for terms related to the graded comonadic type are presented in Figure 2.10. The rules C-Extract, C-Fmap, and C-Fork are generalizations of the corresponding rules for

$\boxed{\Gamma \vdash a : A}$ *(Typing rules)*

$$\text{C-Extract}\ \frac{\Gamma \vdash a : D_1\ A}{\Gamma \vdash \mathbf{extr}\ a : A} \qquad \text{C-Fmap}\ \frac{\Gamma \vdash f : A \to B}{\Gamma \vdash \mathbf{lift}^m f : D_m\ A \to D_m\ B} \qquad \text{C-Fork}\ \frac{\Gamma \vdash a : D_{m_1 \cdot m_2}\ A}{\Gamma \vdash \mathbf{fork}^{m_1,m_2}\ a : D_{m_1}\ D_{m_2}\ A}$$

$$\text{C-Up}\ \frac{\Gamma \vdash a : D_{m_1}\ A \qquad m_1 \leq m_2}{\Gamma \vdash \mathbf{up}^{m_1,m_2}\ a : D_{m_2}\ A}$$

Figure 2.10: Typing rules of GCC (excerpt)

the ungraded comonadic type. The rule C-Up, like rule M-Up, relaxes the grade on the type. If $\mathcal{M}$ is the trivial preordered monoid, then the above rules degenerate to the standard typing rules for comonads. Note that rules C-Fmap and C-Up are essentially the same as rules M-Fmap and M-Up respectively whereas rules C-Extract and C-Fork are like 'inverses' of rules M-Return and M-Join respectively.

The equational theory appears in Figure 2.11 (we omit the $\beta\eta$-rules of standard $\lambda$-calculus). For presenting the rules, we use shorthand notation: $f\ {}^{m_2}\!\ll\!\!\Leftarrow^{m_1}\ a \triangleq (\mathbf{lift}^{m_1} f)(\mathbf{fork}^{m_1,m_2}\ a)$ where $a : D_{m_1 \cdot m_2}\ A$ and $f : D_{m_2}\ A \to B$. Note that $\_\ {}^{m_2}\!\ll\!\!\Leftarrow^{m_1}\ \_$ is a graded-**extend** operator.

The first four rules are same as their counterparts in GMC. The next two rules correspond to commutativity of **extend** with **up**. The two rules after that correspond to **extr** being the left and right identity of **extend**. The last rule corresponds to associativity of **extend**.

$$\mathbf{lift}^m(\lambda x.x) \equiv \lambda x.x \tag{2.10}$$
$$\mathbf{lift}^m(\lambda x.g\ (f\ x)) \equiv \lambda x.(\mathbf{lift}^m g)\ ((\mathbf{lift}^m f)\ x) \tag{2.11}$$
$$\mathbf{up}^{m_1,m_1}\ a \equiv a \tag{2.12}$$
$$\mathbf{up}^{m_2,m_3}(\mathbf{up}^{m_1,m_2}\ a) \equiv \mathbf{up}^{m_1,m_3}\ a \tag{2.13}$$
$$f\ {}^{m_2}\!\ll\!\!\Leftarrow^{m_1'}\ (\mathbf{up}^{m_1 \cdot m_2, m_1' \cdot m_2}\ a) \equiv \mathbf{up}^{m_1,m_1'}(f\ {}^{m_2}\!\ll\!\!\Leftarrow^{m_1}\ a) \tag{2.14}$$
$$f\ {}^{m_2'}\!\ll\!\!\Leftarrow^{m_1}\ (\mathbf{up}^{m_1 \cdot m_2, m_1 \cdot m_2'}\ a) \equiv (\lambda x.f\ (\mathbf{up}^{m_2,m_2'}\ x))\ {}^{m_2}\!\ll\!\!\Leftarrow^{m_1}\ a \tag{2.15}$$
$$\mathbf{extr}\ (f\ {}^{m_2}\!\ll\!\!\Leftarrow^{1}\ a) \equiv f\ a \tag{2.16}$$
$$(\lambda x.\mathbf{extr}\ x)\ {}^{1}\!\ll\!\!\Leftarrow^{m_2}\ a \equiv a \tag{2.17}$$
$$g\ {}^{m_2}\!\ll\!\!\Leftarrow^{m_1}\ (f\ {}^{m_3}\!\ll\!\!\Leftarrow^{m_1 \cdot m_2}\ a) \equiv (\lambda x.g\ (f\ {}^{m_3}\!\ll\!\!\Leftarrow^{m_2}\ x))\ {}^{m_2 \cdot m_3}\!\ll\!\!\Leftarrow^{m_1}\ a \tag{2.18}$$

Figure 2.11: Equality rules of GCC (excerpt)

Next, we want to interpret the calculus in a suitable category. Similar to GMC, the types of standard $\lambda$-calculus can be interpreted in any bicartesian closed category. To interpret the graded comonadic type, we need a strong graded comonad, the dual of a strong graded monad we saw earlier. We briefly go over the definition of a strong graded comonad and present the categorical model thereafter.

### 2.4.2 Graded Comonad and Categorical Model

While a graded monad is a kind of lax monoidal functor, a graded comonad is a kind of oplax monoidal functor. An oplax monoidal functor from a monoidal category $(M, \otimes_M, 1_M)$ to a monoidal category $(N, \otimes_N, 1_N)$ is nothing but a lax monoidal functor from $(M^{\mathrm{op}}, \otimes_M, 1_M)$ to $(N^{\mathrm{op}}, \otimes_N, 1_N)$. Now, given a preordered monoid $\mathcal{M}$, an $\mathcal{M}$-graded comonad over a category $\mathbb{C}$ is an oplax monoidal functor from $\mathbf{C}(\mathcal{M})$ to $\mathbf{End}_{\mathbb{C}}$. In order to interpret GCC, we need to add strength to the graded comonad. We follow the same strategy as before and define a strong $\mathcal{M}$-graded comonad over a cartesian category $\mathbb{C}$ as an oplax monoidal functor from $\mathbf{C}(\mathcal{M})$ to $\mathbf{End}^{s}_{\mathbb{C}}$. We use a strong graded comonad to build the categorical model of GCC.

Let $\mathcal{M} = (M, \_\cdot\_, 1, \leq)$ be the preordered monoid parametrizing the calculus. Let $\mathbb{C}$ be any bicartesian closed category. Let $(\mathbf{D}, \delta, \epsilon)$ be a strong $\mathcal{M}$-graded comonad over $\mathbb{C}$. Then, the interpretation, $[\![\_]\!]$, of types and terms of GCC is as given in Figure 2.12.

$$
\begin{aligned}
[\![D_m\ A]\!] &= \mathbf{D}_m[\![A]\!] & [\![\mathbf{extr}\ a]\!] &= \epsilon \circ [\![a]\!] \\
[\![\mathbf{lift}^m f]\!] &= \Lambda(\mathbf{D}_m(\Lambda^{-1}[\![f]\!]) \circ t^{\mathbf{D}_m}) & [\![\mathbf{fork}^{m_1,m_2}\, a]\!] &= \delta^{m_1,m_2} \circ [\![a]\!] \\
[\![\mathbf{up}^{m_1,m_2}\, a]\!] &= \mathbf{D}^{m_1 \leq m_2} \circ [\![a]\!]
\end{aligned}
$$

FIGURE 2.12: Interpretation of GCC (excerpt)

Note that $\epsilon$ is a natural transformation from $\mathbf{D}_1$ to $\mathbf{Id}$ and $\delta^{m_1,m_2}$ are morphisms from $\mathbf{D}_{m_1 \cdot m_2}$ to $\mathbf{D}_{m_1} \circ \mathbf{D}_{m_2}$, natural in both $m_1$ and $m_2$. By reasoning along the lines of Theorem 2.1, we can show that the above interpretation provides a sound model of GCC.

**Theorem 2.3** If $\Gamma \vdash a : A$ in GCC, then $[\![a]\!] \in \mathrm{Hom}_{\mathbb{C}}([\![\Gamma]\!], [\![A]\!])$. Further, if $\Gamma \vdash a_1 : A$ and $\Gamma \vdash a_2 : A$ such that $a_1 \equiv a_2$ in GCC, then $[\![a_1]\!] = [\![a_2]\!] \in \mathrm{Hom}_{\mathbb{C}}([\![\Gamma]\!], [\![A]\!])$.

Now that we have a graded comonadic calculus with us, we put the comonadic character of DCC to the test. In particular, we test: with appropriate restrictions, can one translate GCC into DCC?

## 2.5 DCC and GCC

In Section 2.3.4, we saw that DCC is not just a graded monadic calculus. The rule Prot-Already lends a comonadic character to it. But does it make DCC a graded comonadic calculus? In other words, over the class of bounded join-semilattices, can we translate GCC into DCC? We can translate $D_\ell\, A$ to $\mathcal{T}_\ell\, A$. The constructs **lift** and **up** can be translated as in Figure 2.9. But to translate **extr** and **fork**, we need to relook at the protection judgment.

### 2.5.1 Protection Judgment, Revisited

To translate **extr** and **fork**, we need to be able to construct functions having types $\mathcal{T}_\bot\, A \to A$ and $\mathcal{T}_{\ell_1 \sqcup \ell_2}\, A \to \mathcal{T}_{\ell_1}\, \mathcal{T}_{\ell_2}\, A$ respectively, for an arbitrary type $A$. However, given the formulation of DCC by Abadi et al. [1999], such a construction is not possible. This is so because in order to construct a function having type $\mathcal{T}_\bot\, A \to A$, we need to show that: $\bot \sqsubseteq A$, for an arbitrary $A$. The protection rules do not allow such a derivation. Similarly, in order to construct a function having type $\mathcal{T}_{\ell_1 \sqcup \ell_2}\, A \to \mathcal{T}_{\ell_1}\, \mathcal{T}_{\ell_2}\, A$, we need to show that: $\ell_1 \sqcup \ell_2 \sqsubseteq \mathcal{T}_{\ell_1}\, \mathcal{T}_{\ell_2}\, A$, for an arbitrary $A$. Again, such a derivation is not allowed by the protection rules.

However, from a dependency perspective, $\bot \sqsubseteq A$ and $\ell_1 \sqcup \ell_2 \sqsubseteq \mathcal{T}_{\ell_1}\, \mathcal{T}_{\ell_2}\, A$ are sound judgments. The judgment $\bot \sqsubseteq A$ is sound because: $\bot$ is the lowest security level and as such, the terms of any type are at least as secure as $\bot$. The judgment $\ell_1 \sqcup \ell_2 \sqsubseteq \mathcal{T}_{\ell_1}\, \mathcal{T}_{\ell_2}\, A$ is sound because: $\mathcal{T}_{\ell_1}\, \mathcal{T}_{\ell_2}\, A$ is at least as secure as $\ell_1$ and $\mathcal{T}_{\ell_1}\, \mathcal{T}_{\ell_2}\, A$ is also at least as secure as $\ell_2$, so it must be at least as secure as $\ell_1 \sqcup \ell_2$. This reasoning is supported by the lattice model of Denning [1976].

Now the above judgments are not only sound, but also desirable. Compared to $A$, the type $\mathcal{T}_\bot\, A$ offers no extra protection. So, it makes sense to allow programs like the one shown below.

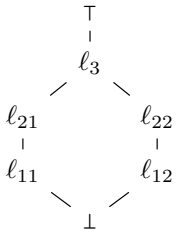
$$x : \mathcal{T}_\bot\, A \vdash \mathbf{bind}^\bot\ y = x\ \mathbf{in}\ y : A$$

Next, the type $\mathcal{T}_{\ell_1}\, \mathcal{T}_{\ell_2}\, A$ offers no less protection than the type $\mathcal{T}_{\ell_1 \sqcup \ell_2}\, A$. So, programs like:

$$x : \mathcal{T}_{\ell_1 \sqcup \ell_2}\, A \vdash \mathbf{bind}^{\ell_1 \sqcup \ell_2}\ y = x\ \mathbf{in}\ \mathbf{eta}^{\ell_1}\mathbf{eta}^{\ell_2}\, y : \mathcal{T}_{\ell_1}\, \mathcal{T}_{\ell_2}\, A$$

should also be allowed.

Allowing programs like the one above has some interesting consequences. For example, consider the lattice shown below.



Here, $\ell_{11} \sqcup \ell_{12} = \ell_3$. Now, putting $\ell_1 := \ell_{11}$ and $\ell_2 := \ell_{12}$ in the above program, we have:

$$x : \mathcal{T}_{\ell_3}\ A \vdash \mathbf{bind}^{\ell_3}\ y = x\ \mathbf{in}\ \mathbf{eta}^{\ell_{11}}\mathbf{eta}^{\ell_{12}}\, y : \mathcal{T}_{\ell_{11}}\ \mathcal{T}_{\ell_{12}}\ A.$$

This program shows that we can observe $\ell_3$-level values in an environment simultaneously protected by $\ell_{11}$ and $\ell_{12}$. Two points are worth noting here.

- First, $\ell_{11}$ and $\ell_{12}$ together offer much much more protection than either of them individually. Observe that neither $\ell_{11}$ nor $\ell_{12}$ individually offer as much protection as either $\ell_{21}$ or $\ell_{22}$. But, $\ell_{11}$ and $\ell_{12}$ together offer more protection than both $\ell_{21}$ and $\ell_{22}$. Behind this observation, lies a fundamental security principle, the principle that forms the basis of applications like two-factor authentication, two-man rule, etc. It may be phrased in terms of the age-old proverb: the whole is more than just the sum of its parts.
- Second, $\ell_3$ is compromised if $\ell_{11}$ and $\ell_{12}$ are simultaneously compromised. This is so because with a simultaneous access to $\ell_{11}$ and $\ell_{12}$, one has access to $\ell_3$ and all levels below it, even $\ell_{21}$ and $\ell_{22}$. It may look counter-intuitive but that just shows the power of simultaneous access and simultaneous protection.

To enable such reasoning within the calculus, we add the following rules to the protection judgment of DCC.

$\boxed{\ell \sqsubseteq A}$ *(Extended Protection Rules)*

$$\textsc{Prot-Minimum}\ \frac{}{\bot \sqsubseteq A} \qquad \textsc{Prot-Combine}\ \frac{\ell_1 \sqsubseteq A \quad \ell_2 \sqsubseteq A \quad \ell \sqsubseteq \ell_1 \sqcup \ell_2}{\ell \sqsubseteq A}$$

As a side note, we shall point out that in the categorical model of DCC given by Abadi et al. [1999], for an arbitrary $A$, the interpretations of $\mathcal{T}_\bot\ A$ and $\mathcal{T}_{\ell_1 \sqcup \ell_2}\ A$ are the same as those of $A$ and $\mathcal{T}_{\ell_1}\ \mathcal{T}_{\ell_2}\ A$ respectively. As such, one can show that DCC extended with the above rules enjoys the same categorical model. We shall, for the sake of precision, call DCC extended with these rules $\text{DCC}_e$.

The equational theory of $\text{DCC}_e$ is defined in the same way as that of DCC. DCC, then, is a proper sub-language of $\text{DCC}_e$. Now, since DCC is a graded monadic calculus, so is $\text{DCC}_e$. However, owing to the reasons described above, DCC is not a graded comonadic calculus. But $\text{DCC}_e$ is a graded comonadic calculus, as we see next.

### 2.5.2 $\mathrm{DCC}_e$ is a Graded Comonadic Calculus

Over the class of bounded join-semilattices, we can translate GCC into $\mathrm{DCC}_e$. Let $\mathcal{L} = (L, \sqcup, \bot)$ be a bounded join-semilattice. Then, the translation $\overline{\ }$ is given in Figure 2.13. Note the role played by rules Prot-Minimum and Prot-Combine in the translation of $\mathbf{extr}\, a$ and $\mathbf{fork}^{\ell_1,\ell_2}\, a$ respectively. The next theorem shows that this translation preserves typing and meaning.

$$
\begin{aligned}
\overline{D_\ell\ A} &= \mathcal{T}_\ell\ \overline{A} \\
\overline{\mathbf{extr}\ a} &= \mathbf{bind}^{\bot}\ x = \overline{a}\ \mathbf{in}\ x \\
\overline{\mathbf{lift}^{\ell} f} &= \lambda x : \mathcal{T}_\ell\ \overline{A}.\mathbf{bind}^{\ell}\ y = x\ \mathbf{in}\ \mathbf{eta}^{\ell}(\overline{f}\ y)\ \text{[Here, } f : A \to B\text{]} \\
\overline{\mathbf{fork}^{\ell_1,\ell_2}\, a} &= \mathbf{bind}^{\ell_1 \sqcup \ell_2}\ x = \overline{a}\ \mathbf{in}\ \mathbf{eta}^{\ell_1}\mathbf{eta}^{\ell_2} x \\
\overline{\mathbf{up}^{\ell_1,\ell_2}\, a} &= \mathbf{bind}^{\ell_1}\ x = \overline{a}\ \mathbf{in}\ \mathbf{eta}^{\ell_2} x
\end{aligned}
$$

Figure 2.13: Translation function from GCC to $\mathrm{DCC}_e$ (excerpt)

**Theorem 2.4** If $\Gamma \vdash a : A$ in $\mathrm{GCC}(\mathcal{L})$, then $\overline{\Gamma} \vdash \overline{a} : \overline{A}$ in $\mathrm{DCC}_e(\mathcal{L})$. Further, if $\Gamma \vdash a_1 : A$ and $\Gamma \vdash a_2 : A$ such that $a_1 \equiv a_2$ in $\mathrm{GCC}(\mathcal{L})$, then $\overline{a_1} \simeq \overline{a_2}$ in $\mathrm{DCC}_e(\mathcal{L})$.

To summarize, earlier we showed that DCC is a graded monadic calculus by translating GMC into it. However, we couldn't translate DCC into GMC because DCC has some comonadic character to it. Thereafter, we designed a graded comonadic calculus, GCC. Using GCC, we put the comonadic character of DCC to the test. We could not translate GCC to DCC, thus showing that DCC is not fully comonadic. Next, we found that such a translation is not possible only because DCC does not allow certain derivations that are both sound and desirable. We extended DCC to $\mathrm{DCC}_e$ to allow these derivations and found that we can translate GCC (and GMC) into $\mathrm{DCC}_e$. Now, can we go the other way around and translate $\mathrm{DCC}_e$ into a calculus built up using just GMC and GCC?

## 2.6 Graded Monadic Comonadic Calculus

### 2.6.1 The Calculus

The Graded Monadic Comonadic Calculus (GMCC) combines the Graded Monadic Calculus (GMC) and the Graded Comonadic Calculus (GCC) into a single system. We can view it as an extension of the standard simply-typed $\lambda$-calculus with a graded type constructor $S_m$, which behaves both like a graded monadic type constructor, $T_m$, and a graded comonadic type constructor, $D_m$. The calculus has as terms the union of those of GMC and GCC. The typing rules of the calculus include the rules of GMC and GCC (shown in Figures 2.2 and 2.10 respectively) with $S_m$ replacing $T_m$ and $D_m$.

The equational theory of the calculus is generated by the equational theories of GMC and GCC (presented in Figures 2.3 and 2.11 respectively) along with the following additional rules: $\mathbf{extr}(\mathbf{ret}\, a) \equiv a$ and $\mathbf{ret}(\mathbf{extr}\, a) \equiv a$ and $\mathbf{fork}^{m_1,m_2}(\mathbf{join}^{m_1,m_2}\, a) \equiv a$ and $\mathbf{join}^{m_1,m_2}(\mathbf{fork}^{m_1,m_2}\, a) \equiv a$. These additional rules ensure that the monadic and the comonadic fragments of the calculus behave well with respect to one another.

### 2.6.2 Categorical Model

GMCC enjoys a nice categorical model, as we show next.
We interpret the graded type constructor of the calculus as a kind of strong monoidal functor [MacLane, 1971]. A strong monoidal functor from a monoidal category $(M, \otimes_M, 1_M)$ to a monoidal category $(N, \otimes_N, 1_N)$ is a lax monoidal functor $(F, F_2, F_0) : (M, \otimes_M, 1_M) \to (N, \otimes_N, 1_N)$ where $F_0$ and $F_2(X, Y)$ are invertible for all $X, Y \in \mathrm{Obj}(M)$. Thus, for a strong monoidal functor $S : (M, \otimes_M, 1_M) \to (N, \otimes_N, 1_N)$, we have: $S(1_M) \cong 1_N$ and $S(X \otimes_M Y) \cong S(X) \otimes_N S(Y)$ for all $X, Y \in \mathrm{Obj}(M)$. Note that if these isomorphisms are identities, then the functor is said to be strict. Further, note that the word 'strong' in 'strong monoidal functor' and in 'strong endofunctor' refer to different properties.

Let $\mathcal{M} = (M, \_ \cdot \_, 1, \leq)$ be the preordered monoid parametrizing the calculus. Let $\mathbb{C}$ be any bicartesian closed category. Let $\mathbf{S}$ be a strong monoidal functor from $\mathbf{C}(\mathcal{M})$ to $\mathbf{End}^s_{\mathbb{C}}$. Then $\mathbf{S}$ is both a strong $\mathcal{M}$-graded monad over $\mathbb{C}$ and a strong $\mathcal{M}$-graded comonad over $\mathbb{C}$. With respect to $\mathbf{S}$, let $\mu, \eta, \delta, \epsilon$ denote the corresponding natural transformations. Then,

$$\epsilon \circ \eta = \mathrm{id} \qquad \eta \circ \epsilon = \mathrm{id}$$
$$\delta \circ \mu = \mathrm{id} \qquad \mu \circ \delta = \mathrm{id}$$

We interpret $S_m\ A$ as: $[\![S_m\ A]\!] = \mathbf{S}_m [\![A]\!]$. The terms are interpreted as in Figures 2.7 and 2.12. This gives us a sound interpretation of the calculus.

**Theorem 2.5** If $\Gamma \vdash a : A$ in GMCC, then $[\![a]\!] \in \mathrm{Hom}_{\mathbb{C}}([\![\Gamma]\!], [\![A]\!])$. Further, if $\Gamma \vdash a_1 : A$ and $\Gamma \vdash a_2 : A$ such that $a_1 \equiv a_2$ in GMCC, then $[\![a_1]\!] = [\![a_2]\!] \in \mathrm{Hom}_{\mathbb{C}}([\![\Gamma]\!], [\![A]\!])$.

The theorem above shows that given a preordered monoid $\mathcal{M}$, any bicartesian closed category $\mathbb{C}$ together with a strong monoidal functor from $\mathbf{C}(\mathcal{M})$ to $\mathbf{End}^s_{\mathbb{C}}$ provides a sound model of GMCC($\mathcal{M}$). In addition to soundness, GMCC($\mathcal{M}$) also enjoys completeness with respect to its class of categorical models. Formally, we can show:

**Theorem 2.6** Given any preordered monoid $\mathcal{M}$, for typing derivations $\Gamma \vdash a_1 : A$ and $\Gamma \vdash a_2 : A$ in GMCC($\mathcal{M}$), if $[\![a_1]\!] = [\![a_2]\!]$ in all models of GMCC($\mathcal{M}$), then $a_1 \equiv a_2$ is derivable in GMCC($\mathcal{M}$).

We use entirely standard term-model techniques [Jacobs, 1999] to prove completeness. First, we construct the classifying category and thereafter, the generic model in the classifying category. The generic model equates only the terms that are equal in the calculus. So, if the interpretations

of two GMCC($\mathcal{M}$)-terms are equal in all models (and therefore in the generic model too), then these terms are equal in the calculus as well. Further, we can show that:

**Theorem 2.7** The generic model satisfies the universal property.

The above theorem implies that any model of GMCC($\mathcal{M}$) can be factored through the generic model. We don't explore the consequences of this theorem here, but leave it for future work.

In this section, we have seen that GMCC is sound and complete with respect to its class of categorical models. In the next section, we explore the relation between GMCC and $\mathrm{DCC}_e$.

## 2.7 GMCC and $\mathrm{DCC}_e$

We saw that over the class of bounded join-semilattices, we can translate both GMC and GCC into $\mathrm{DCC}_e$. In fact, over the same class of structures, we can go a bit further and translate GMCC into $\mathrm{DCC}_e$, following the translations presented in Figures 2.9 and 2.13.

**Theorem 2.8** If $\Gamma \vdash a : A$ in GMCC($\mathcal{L}$), then $\overline{\Gamma} \vdash \overline{a} : \overline{A}$ in $\mathrm{DCC}_e(\mathcal{L})$. Further, if $\Gamma \vdash a_1 : A$ and $\Gamma \vdash a_2 : A$ such that $a_1 \equiv a_2$ in GMCC($\mathcal{L}$), then $\overline{a_1} \simeq \overline{a_2}$ in $\mathrm{DCC}_e(\mathcal{L})$.

We now go the other way around and translate $\mathrm{DCC}_e(\mathcal{L})$ into GMCC($\mathcal{L}$).

Note here that $\mathrm{DCC}_e$ has a very liberal definition of equality, inherited from DCC. Two $\mathrm{DCC}_e$-terms are equal if, after erasure, they are equal as $\lambda$-terms. So $\mathbf{eta}^{\ell}\, a \simeq a$ for any $\mathrm{DCC}_e$-term $a$. The same is not true in general in GMCC. For example, $\mathbf{up}^{\bot,\ell}(\mathbf{ret}\ a)$ may not be equal to $a$. To capture the notion of $\mathrm{DCC}_e$-equality in GMCC, we need to define a similar relation between GMCC terms. For GMCC terms $a_1$ and $a_2$, we say $a_1 \simeq a_2$ if and only if $\lfloor a_1 \rfloor$ and $\lfloor a_2 \rfloor$, the plain $\lambda$-term counterparts of $a_1$ and $a_2$ respectively, are equal as $\lambda$-terms. The erasure operation $\lfloor\ \rfloor$ strips away the constructors $\mathbf{ret}, \mathbf{join}, \mathbf{extr}, \mathbf{fork}, \mathbf{lift}$ and $\mathbf{up}$ along with the grade annotations. For example, $\lfloor \mathbf{join}^{\ell_1,\ell_2}\, a \rfloor = \lfloor a \rfloor$.

Coming back to the translation from $\mathrm{DCC}_e$ to GMCC, it is straightforward for types. The modal type $\mathcal{T}_\ell\, B$ gets translated to $S_\ell\, \underline{B}$. We use $\_$ to denote the translation function. For translating terms, we need the following lemma. Note that in Section 2.3.4, while trying to translate DCC to GMC, we could not prove a lemma like this one.

**Lemma 2.9** If $\ell \sqsubseteq B$, then there exists a term $\varnothing \vdash j : S_\ell\, \underline{B} \to \underline{B}$ such that $\lfloor j \rfloor \equiv \lambda x.x$.

Now, with the above lemma, we can translate $\mathrm{DCC}_e$ terms to GMCC terms, as shown below. The function $j$ used in the translation is as given by Lemma 2.9.

$$\underline{\mathbf{eta}^{\ell}\, a} = \mathbf{up}^{\bot,\ell}(\mathbf{ret}\ \underline{a}) \qquad \underline{\mathbf{bind}^{\ell}\ x = a\ \mathbf{in}\ b} = j\ ((\mathbf{lift}^{\ell}(\lambda x.\underline{b}))\ \underline{a})$$

This translation preserves typing and meaning.

**Theorem 2.10** If $\Gamma \vdash a : A$ in $\mathrm{DCC}_e(\mathcal{L})$, then $\underline{\Gamma} \vdash \underline{a} : \underline{A}$ in $\mathrm{GMCC}(\mathcal{L})$. Further, if $\Gamma \vdash a_1 : A$ and $\Gamma \vdash a_2 : A$ such that $a_1 \simeq a_2$ in $\mathrm{DCC}_e(\mathcal{L})$, then $\underline{a_1} \simeq \underline{a_2}$ in $\mathrm{GMCC}(\mathcal{L})$.

Theorems 2.10 and 2.8 together show that over the class of bounded join-semilattices, GMCC and $\mathrm{DCC}_e$, seen as dependency calculi, are equivalent. Thus, GMCC is a generalization of the Dependency Core Calculus, DCC. Hence, dependency analysis, at least to the extent DCC is capable of, can be done using just a graded monadic comonadic calculus. This connection between dependency analysis and GMCC is important because:

- Dependency analysis can now benefit from a wider variety of categorical models. Some of these models may provide simpler proofs of correctness. In fact, using our categorical models, we show, in a straightforward manner, that dependency analyses in $\mathrm{DCC}_e$ and $\lambda^\circ$ are correct.
- More dependency calculi can now be unified under a common framework. As a proof of concept, we show that the binding-time analyzing calculus $\lambda^\circ$ of Davies [2017] can be encoded into $\mathrm{GMCC}_e$, an extension of GMCC. Note that $\lambda^\circ$ can not be translated into DCC [Abadi et al., 1999].
- The non-standard **bind**-rule of DCC can be replaced with standard monadic and comonadic typing rules. This finding provides insight into the categorical basis of the **bind**-rule of DCC. (See Section 2.8.1)
- GMCC is formed combining GMC and its dual, GCC. GMC can be seen as a restriction of the Explicit Subeffecting Calculus of Katsumata [2014]. The Explicit Subeffecting Calculus is a general system for analyzing effects. This clean connection between dependency analysis and effect analysis promises to be a fertile ground for new ideas, especially in the intersection of dependency, effect and coeffect analyses.

Before closing this section, we want to make some remarks on the equivalence between GMCC and $\mathrm{DCC}_e$. From theorems 2.10 and 2.8, we see that GMCC and $\mathrm{DCC}_e$ are equivalent upto erasure. We may wonder: can this equivalence be made stronger? The answer is yes. In the next section, we shall show that (over the class of bounded join-semilattices) GMCC and $\mathrm{DCC}_e$ are semantically equivalent too. Further, we can show something stronger on the syntactic side as well:

**Theorem 2.11** Let $\Gamma \vdash a : A$ be any derivation in $\mathrm{GMCC}(\mathcal{L})$. Then, $\overline{\underline{a}} \equiv a$.

The above theorem says that any $\mathrm{GMCC}(\mathcal{L})$-term, after a round trip to $\mathrm{DCC}_e(\mathcal{L})$, is equal (not just equal upto erasure) to itself. When going the other way, we can only prove the following weaker result: if $\Gamma \vdash a : A$ is a derivation in $\mathrm{DCC}_e(\mathcal{L})$, then $\overline{\underline{a}} \simeq a$. Note that we are forced to use equality upto erasure here because we don't have an alternative equational theory for $\mathrm{DCC}_e$.

Next, we develop the categorical semantics for $\mathrm{DCC}_e$.

## 2.8 $\mathrm{DCC}_e$: Categorical Semantics

Abadi et al. [1999] provide a categorical model for DCC and prove noninterference using that model. Several other authors [Algehed and Bernardy, 2019, Bowman and Ahmed, 2015, Shikuma and Igarashi, 2006, Tse and Zdancewic, 2004] have provided alternative proofs of noninterference for DCC using various techniques, including parametricity. In this section, we present a class of categorical models for $\mathrm{DCC}_e$, in the style of GMCC, and show noninterference for the calculus using these models. We also establish semantic equivalence between $\mathrm{DCC}_e$ and GMCC and explain the non-standard **bind**-rule of DCC in terms of standard category-theoretic concepts.

### 2.8.1 Categorical Models for $\mathrm{DCC}_e$

Given that GMCC and $\mathrm{DCC}_e$, considered as dependency calculi, are equivalent, we can simply use models of GMCC to interpret $\mathrm{DCC}_e$. However, in this section, we build models for $\mathrm{DCC}_e$ from first principles and show them to be computationally adequate with respect to a call-by-value semantics. The original operational semantics of DCC, presented by Abadi et al. [1999], is somewhat ad hoc from a categorical perspective because according to this semantics, $\mathbf{eta}^\ell\, a$ converts to $a$. If we interpret **eta** as the unit of a monad, this conversion would require us to interpret the unit as the identity natural transformation, something that is not very general. On the other hand, a call-by-value semantics or a call-by-name semantics of the calculus [Tse and Zdancewic, 2004] is quite general from a category theoretic perspective. According to a call-by-value semantics, **bind**-expressions convert in the following manner:

$\boxed{\vdash a \rightsquigarrow a'}$ *(Operational Semantics)*

$$\textsc{CBV-BindLeft}\ \frac{\vdash a \rightsquigarrow a'}{\vdash \mathbf{bind}^\ell\ x = a\ \mathbf{in}\ b \rightsquigarrow \mathbf{bind}^\ell\ x = a'\ \mathbf{in}\ b} \qquad \textsc{CBV-BindBeta}\ \frac{}{\vdash \mathbf{bind}^\ell\ x = \mathbf{eta}^\ell\, v\ \mathbf{in}\ b \rightsquigarrow b\{v/x\}}$$

Given $\rightsquigarrow$, we can define the multistep reduction relation, $\rightsquigarrow^*$, in the usual way. A point to note here is that though we use a call-by-value semantics for $\mathrm{DCC}_e$, we could have used a call-by-name semantics as well. $\mathrm{DCC}_e$ being a terminating calculus, the choice of one evaluation strategy over the other does not lead to significant differences in metatheory. We chose call-by-value over call-by-name because the former allows reductions underneath the **eta**s.

Given a bounded join-semilattice, $\mathcal{L} = (L, \sqcup, \bot)$, and any bicartesian closed category $\mathbb{C}$, we interpret the graded type constructor $\mathcal{T}$ of $\mathrm{DCC}_e(\mathcal{L})$ as a strong monoidal functor $\mathbf{S}$ from $\mathbf{C}(\mathcal{L})$ to $\mathbf{End}^{\mathrm{s}}_{\mathbb{C}}$. Formally, $[\![\mathcal{T}_\ell\, A]\!] = \mathbf{S}_\ell[\![A]\!]$. Note that since $\sqcup$ is idempotent, the triple $(\mathbf{S}_\ell, \mathbf{S}^{\bot \sqsubseteq \ell} \circ \eta, \mu^{\ell,\ell})$, for any $\ell \in L$, is a monad. Further, since $\mu^{\ell,\ell}$ is invertible, such a monad is also idempotent.

For interpreting terms, we need the following lemma.

**Lemma 2.12** If $\ell \sqsubseteq A$, then $\exists$ an isomorphism $k : \mathbf{S}_\ell[\![A]\!] \to [\![A]\!]$ such that $k^{-1} = \mathbf{S}^{\bot \sqsubseteq \ell} \circ \eta$.

Using the above lemma, we interpret terms as:

$$[\![\mathbf{eta}^\ell a]\!] = \mathbf{S}^{\bot \sqsubseteq \ell} \circ \eta \circ [\![a]\!]$$
$$[\![\mathbf{bind}^\ell\ x = a\ \mathbf{in}\ b]\!] = k \circ \mathbf{S}_\ell[\![b]\!] \circ \mathrm{t}^{\mathbf{S}_\ell} \circ \langle \mathrm{id}, [\![a]\!] \rangle$$

For later reference, note that whenever we need to be precise, we use $[\![\_]\!]_{(\mathbb{C},\mathbf{S})}$ to refer to the interpretation of $\mathrm{DCC}_e(\mathcal{L})$ in category $\mathbb{C}$ using $\mathbf{S}$. The above interpretation of $\mathrm{DCC}_e$ is sound, as we see next.

**Theorem 2.13** If $\Gamma \vdash a : A$ in $\mathrm{DCC}_e$, then $[\![a]\!] \in \mathrm{Hom}_{\mathbb{C}}([\![\Gamma]\!], [\![A]\!])$.

**Theorem 2.14** If $\Gamma \vdash a : A$ in $\mathrm{DCC}_e$ and $\vdash a \rightsquigarrow a'$, then $[\![a]\!] = [\![a']\!]$.

Computational adequacy with respect to call-by-value operational semantics follows as a corollary of the above theorem. Below, we assume that the categorical interpretation is injective for ground types. In particular, $[\![\mathbf{true}]\!] \neq [\![\mathbf{false}]\!]$, where $\mathbf{true} \triangleq \mathbf{inj}_1\,\mathbf{unit} : \mathbf{Bool}$ and $\mathbf{false} \triangleq \mathbf{inj}_2\,\mathbf{unit} : \mathbf{Bool}$, and $\mathbf{Bool} \triangleq \mathbf{Unit} + \mathbf{Unit}$.

**Theorem 2.15** Let the interpretation $[\![\_]\!]_{(\mathbb{C},\mathbf{S})}$ be injective for ground types.

- Let $\Gamma \vdash b : \mathbf{Bool}$ and $v$ be a value of type $\mathbf{Bool}$. If $[\![b]\!]_{(\mathbb{C},\mathbf{S})} = [\![v]\!]_{(\mathbb{C},\mathbf{S})}$, then $\vdash b \rightsquigarrow^* v$.
- Fix some $\ell \in L$. Let $\Gamma \vdash b : \mathcal{T}_\ell\,\mathbf{Bool}$ and $v$ be a value of type $\mathcal{T}_\ell\,\mathbf{Bool}$. Suppose, the morphisms $\overline{\eta}_X \triangleq \mathbf{S}_X^{\bot \sqsubseteq \ell} \circ \eta_X$ are mono for any $X \in \mathrm{Obj}(\mathbb{C})$. Now, if $[\![b]\!]_{(\mathbb{C},\mathbf{S})} = [\![v]\!]_{(\mathbb{C},\mathbf{S})}$, then $\vdash b \rightsquigarrow^* v$.

Now that $\mathrm{GMCC}(\mathcal{L})$ and $\mathrm{DCC}_e(\mathcal{L})$ enjoy the same class of models, we can show the two calculi are exactly equivalent over these models.

**Theorem 2.16** Let $\mathcal{L}$ be a bounded join-semilattice.

- If $\Gamma \vdash a : A$ in $\mathrm{GMCC}(\mathcal{L})$, then $[\![\overline{a}]\!] = [\![a]\!]$.
- If $\Gamma \vdash a : A$ in $\mathrm{DCC}_e(\mathcal{L})$, then $[\![\underline{a}]\!] = [\![a]\!]$.

The categorical semantics of $\mathrm{DCC}_e$ help in understanding the non-standard **bind**-rule of DCC. By lemma 2.12, if $\ell \sqsubseteq A$, then $\mathbf{S}_\ell[\![A]\!] \cong [\![A]\!]$. In fact, if $\ell \sqsubseteq A$, then $[\![A]\!]$ is the carrier of an $\mathbf{S}_\ell$-algebra. What this means is that the protection judgment is a syntactic mechanism for picking out the appropriate monad algebras. This insight explains the signature of rule DCC-Bind: $\mathcal{T}_\ell\ A \to (A \to B) \to \{\ell \sqsubseteq B\} \to B$. If $\ell \sqsubseteq B$, then $[\![B]\!]$ is the carrier of an $\mathbf{S}_\ell$-algebra. As such, the return type of the rule can be $B$, in lieu of $\mathcal{T}_\ell\ B$. The following theorem characterizes the protection judgment in terms of monad algebras.

**Theorem 2.17** If $\ell \sqsubseteq A$ in $\mathrm{DCC}_e$, then $([\![A]\!], k)$ is an $\mathbf{S}_\ell$-algebra.
Further, if $\ell \sqsubseteq A$ and $\ell \sqsubseteq B$, then for any $f \in \mathrm{Hom}_{\mathbb{C}}([\![A]\!], [\![B]\!])$, $f$ is an $\mathbf{S}_\ell$-algebra morphism.

Hence, the full subcategory of $\mathbb{C}$ with Obj $\coloneqq \{[\![A]\!] \mid \ell \sqsubseteq A\}$ is also a full subcategory of the Eilenberg-Moore category, $\mathbb{C}^{\mathbf{S}_\ell}$.

Next, we use these models to prove noninterference for $\mathrm{DCC}_e$.

### 2.8.2 Proof of Noninterference

Two functors in $\mathbf{End}^{\mathrm{s}}_{\mathbb{C}}$ are crucial to our proof of non-interference. One of them is the identity functor, **Id**. The other is the terminal functor, denoted by $*$, the functor which maps all objects to $\top$, the terminal object of the category and all morphisms to $\langle\rangle$. Now, for every $\ell \in L$, we define a strong monoidal functor $\mathbf{S}^\ell$ from $\mathbf{C}(\mathcal{L})$ to $\mathbf{End}^{\mathrm{s}}_{\mathbb{C}}$ as follows.

$$\mathbf{S}^\ell(\ell') = \begin{cases} \mathbf{Id}, & \text{if } \ell' \sqsubseteq \ell \\ *, & \text{otherwise} \end{cases} \qquad \mathbf{S}^\ell(\ell_1 \sqsubseteq \ell_2) = \begin{cases} \mathrm{id}, & \text{if } \ell_2 \sqsubseteq \ell \\ \langle\rangle, & \text{otherwise} \end{cases}$$

The following points are worth noting here.

- $\mathbf{S}^\ell(\bot) = \mathbf{Id}$ for any $\ell \in L$. Then, for every $\mathbf{S}^\ell$, $\eta = \epsilon = \mathrm{id}$ .
- Fix some $\ell_0 \in L$. Now, for any $\ell_1, \ell_2 \in L$, we have, $\mathbf{S}^{\ell_0}(\ell_1) \circ \mathbf{S}^{\ell_0}(\ell_2) = \mathbf{S}^{\ell_0}(\ell_1 \sqcup \ell_2)$. Then, $\mu^{\ell_1,\ell_2} = \delta^{\ell_1,\ell_2} = \mathrm{id}$. Therefore, the $\mathbf{S}^\ell$s are all strict monoidal functors.

Now, for any bicartesian closed category $\mathbb{C}$, any strong monoidal functor from $\mathbf{C}(\mathcal{L})$ to $\mathbf{End}^{\mathrm{s}}_{\mathbb{C}}$ provides a computationally adequate interpretation of $\mathrm{DCC}_e(\mathcal{L})$, provided the interpretation for ground types is injective. Therefore,

**Theorem 2.18** $[\![\_]\!]_{(\mathbf{Set},\mathbf{S}^\ell)}$, for any $\ell \in L$, is a computationally adequate interpretation of $\mathrm{DCC}_e(\mathcal{L})$.

Next, we explain the intuition behind these strong monoidal functors. $\mathbf{S}^\ell$ keeps untouched all information at levels $\ell'$ where $\ell' \sqsubseteq \ell$ but blacks out all information at every other level. So, $\mathbf{S}^\ell$ corresponds to the view of an observer at level $\ell$. We can formalize this intuition. Suppose $\neg(\ell'' \sqsubseteq \ell)$, i.e. $\ell$ should not depend upon $\ell''$. Then, if $\ell'' \sqsubseteq A$, the terms of type $A$ should not be visible to an observer at level $\ell$. In other words, if $\ell'' \sqsubseteq A$, $\mathbf{S}^\ell$ should black out all information from type $A$. This is indeed the case, as we see next.

**Lemma 2.19** If $\ell'' \sqsubseteq A$ and $\neg(\ell'' \sqsubseteq \ell)$, then $[\![A]\!]_{(\mathbb{C},\mathbf{S}^\ell)} \cong \top$.

The above lemma takes us to our noninterference theorem. Recall the test of correctness for DCC from Section 2.3.1: if $\ell \sqsubseteq A$ and $\neg(\ell \sqsubseteq \ell')$, then the terms of type $A$ should not be visible at $\ell'$. We prove correctness for $\mathrm{DCC}_e$ by formulating this test as:

**Theorem 2.20** Let $\mathcal{L} = (L, \sqcup, \bot)$ be the parametrizing semilattice.

- Suppose $\ell \in L$ such that $\neg(\ell \sqsubseteq \bot)$. Let $\ell \sqsubseteq A$. Let $\varnothing \vdash f : A \to \mathbf{Bool}$ and $\varnothing \vdash a_1 : A$ and $\varnothing \vdash a_2 : A$. Then, $\vdash f\ a_1 \rightsquigarrow^* v$ if and only if $\vdash f\ a_2 \rightsquigarrow^* v$, where $v$ is a value of type $\mathbf{Bool}$.

- Suppose $\ell, \ell' \in L$ such that $\neg(\ell \sqsubseteq \ell')$. Let $\ell \sqsubseteq A$. Let $\varnothing \vdash f : A \to \mathcal{T}_{\ell'}\ \mathbf{Bool}$ and $\varnothing \vdash a_1 : A$ and $\varnothing \vdash a_2 : A$. Then, $\vdash f\ a_1 \rightsquigarrow^* v$ if and only if $\vdash f\ a_2 \rightsquigarrow^* v$, where $v$ is a value of type $\mathcal{T}_{\ell'}\ \mathbf{Bool}$.

As corollary of the above theorem, we can show that $\varnothing \vdash f : \mathcal{T}_{\mathbf{High}}\ A_1 \to \mathbf{Bool}$ and $\varnothing \vdash f' : (A_1 \to \mathcal{T}_{\mathbf{High}}\ A_2) \to \mathbf{Bool}$, for all types $A_1$ and $A_2$, are constant functions. We can also show that $\varnothing \vdash f'' : \mathcal{T}_{\ell_1}\ A \to \mathcal{T}_{\ell_2}\ \mathbf{Bool}$, for any type $A$, is a constant function, whenever $\neg(\ell_1 \sqsubseteq \ell_2)$.

We use the $\mathbf{S}^{\ell}$s to prove the noninterference theorem above. This technique relies on the observation that for two entities $E_1$ and $E_2$, if $E_1$ can be present when $E_2$ is absent, then $E_1$ *does not depend* upon $E_2$. We call this technique the *presence-absence test*. In the next section, we shall use the same technique to prove correctness of binding-time analysis in $\lambda^\circ$.

## 2.9 Binding-Time Calculus, $\lambda^\circ$

$\lambda^\circ$ [Davies, 2017] is a foundational calculus for binding-time analysis, lying at the heart of state-of-the-art metaprogramming languages like MetaOCaml [Calcagno et al., 2003]. $\lambda^\circ$ is essentially a dependency calculus that ensures early stage computations do not depend upon later stage ones. One might expect that DCC, being a core calculus of dependency, would subsume $\lambda^\circ$. However, Abadi et al. [1999] noted that $\lambda^\circ$ cannot be translated into DCC. One reason behind this shortcoming is that DCC does not fully utilize the power of comonadic aspect of dependency analysis, as we discussed before. We extended DCC to $\mathrm{DCC}_e$ to include the comonadic aspect of dependency analysis. This extension opens up the possibility of $\lambda^\circ$ being translated to $\mathrm{DCC}_e$. In this section, we explore this possibility. We first review the calculus $\lambda^\circ$, thereafter present a categorical model leading to an alternative proof of correctness and finally show how we can translate $\lambda^\circ$ into our graded monadic comonadic system.

### 2.9.1 The Calculus $\lambda^\circ$

$\lambda^\circ$ is simply-typed $\lambda$-calculus extended with a 'next time' type constructor, $\bigcirc$. Intuitively, $\bigcirc A$ is the type of terms to be computed upon the 'next time'. The calculus models staged computation, with an earlier stage manipulating programs from later stage as data. For a time-ordered normalization, the calculus needs to ensure that computation from an earlier stage *does not depend* upon computation from a later stage. To model such a notion of independence of the past from the future, Davies [2017] uses temporal logic. In $\lambda^\circ$, time is discretized as instants or moments, represented by natural numbers. For example, 0 denotes the present moment, $0'$

$$\begin{aligned}
&\text{Types, } A, B ::= K \mid A \to B \mid \bigcirc A \mid \mathbf{Unit} \mid A_1 + A_2 \\
&\text{Terms, } a, b ::= x \mid \lambda x : A.b \mid b\ a \mid \mathbf{next}\ a \mid \mathbf{prev}\ a \mid \mathbf{unit} \mid \mathbf{inj}_1\ a_1 \mid \mathbf{inj}_2\ a_2 \mid \mathbf{case}\ a\ \mathbf{of}\ b_1\ ;\ b_2 \\
&\text{Contexts, } \Gamma ::= \varnothing \mid \Gamma, x :^n A
\end{aligned}$$

FIGURE 2.14: Grammar of $\lambda^\circ$

$\boxed{\Gamma \vdash a :^n A}$ *(Typing)*

$$\text{LC-VAR}\ \frac{}{\Gamma_1, x :^n A, \Gamma_2 \vdash x :^n A} \qquad \text{LC-LAM}\ \frac{\Gamma, x :^n A \vdash b :^n B}{\Gamma \vdash \lambda x : A.b :^n A \to B} \qquad \text{LC-APP}\ \frac{\Gamma \vdash b :^n A \to B \quad \Gamma \vdash a :^n A}{\Gamma \vdash b\ a :^n B}$$

$$\text{LC-INJ1}\ \frac{\Gamma \vdash a_1 :^n A_1}{\Gamma \vdash \mathbf{inj}_1\ a_1 :^n A_1 + A_2} \qquad \text{LC-CASE}\ \frac{\Gamma \vdash a :^n A_1 + A_2 \quad \Gamma \vdash b_1 :^n A_1 \to B \quad \Gamma \vdash b_2 :^n A_2 \to B}{\Gamma \vdash \mathbf{case}\ a\ \mathbf{of}\ b_1\ ;\ b_2 :^n B}$$

$$\text{LC-NEXT}\ \frac{\Gamma \vdash a :^{n+0'} A}{\Gamma \vdash \mathbf{next}\ a :^n \bigcirc A} \qquad \text{LC-PREV}\ \frac{\Gamma \vdash a :^n \bigcirc A}{\Gamma \vdash \mathbf{prev}\ a :^{n+0'} A}$$

FIGURE 2.15: Typing Rules of $\lambda^\circ$ (Excerpt)

$$\begin{aligned}
(\lambda x : A.b)\ a &\equiv b\{a/x\} & b &\equiv \lambda x : A.b\ x \\
\mathbf{prev}\ (\mathbf{next}\ a) &\equiv a & a &\equiv \mathbf{next}\ (\mathbf{prev}\ a) \\
\mathbf{case}\ (\mathbf{inj}_i\ a_i)\ \mathbf{of}\ b_1\ ;\ b_2 &\equiv b_i\ a_i & b\ a &\equiv \mathbf{case}\ a\ \mathbf{of}\ \lambda x_1.b\ (\mathbf{inj}_1\ x_1)\ ;\ \lambda x_2.b\ (\mathbf{inj}_2\ x_2)
\end{aligned}$$

FIGURE 2.16: Equality rules of $\lambda^\circ$ (Excerpt)

denotes the next moment and so on. We now look at the calculus formally, as presented by Davies [2017].

The grammar of $\lambda^\circ$ appears in Figure 2.14, typing rules in Figure 2.15 and the equational theory in Figure 2.16. The typing judgment $\Gamma \vdash a :^n A$ intuitively means that $a$ is available at time instant $n$, provided the variables in $\Gamma$ are available at their respective time instants. Note that $\lambda^\circ$ does not have sum types; we include them here for the sake of having non-trivial ground types.

With this background on $\lambda^\circ$, let us now build categorical models for the calculus.

### 2.9.2 Categorical Models for $\lambda^\circ$

The motivation for categorical models of $\lambda^\circ$, in the style of GMCC, comes from the observation that rules LC-NEXT and LC-PREV are like rules C-FORK and M-JOIN respectively. Here, we can think of $\mathcal{N} = (\mathbb{N}, +, 0)$ with discrete ordering to be the parametrizing preordered monoid. Then, $\bigcirc$ is like $S_{0'}$, where $S$ is the graded modal type constructor. We see that $\lambda^\circ$ and GMCC($\mathcal{N}$) share

several similarities, but there is a crucial difference between the two calculi. In $\lambda^\circ$, the types $\bigcirc^n(A \to B)$ and $\bigcirc^n A \to \bigcirc^n B$ (where $\bigcirc^n$ is the operator $\bigcirc$ applied $n$ times) are isomorphic whereas in GMCC($\mathcal{N}$), the types $S_n\,(A \to B)$ and $S_n\,A \to S_n\,B$ are not necessarily isomorphic. Owing to this difference, we need to modify our models in order to interpret $\lambda^\circ$. More precisely, unlike $S_n$, we cannot model $\bigcirc^n$ using any strong endofunctor, but require cartesian closed endofunctors. So next, we define the category of cartesian closed endofunctors.

Let $\mathbb{C}$ be a cartesian closed category. An endofunctor $F : \mathbb{C} \to \mathbb{C}$ is said to be finite-product-preserving if and only if the morphisms $p_\top \triangleq \langle\rangle : F(\top) \to \top$ and $p_{X,Y} \triangleq \langle F\pi_1, F\pi_2\rangle : F(X \times Y) \to FX \times FY$, for $X, Y \in \mathrm{Obj}(\mathbb{C})$, have inverses. A finite-product-preserving endofunctor $F : \mathbb{C} \to \mathbb{C}$ is said to be cartesian closed if and only if the morphisms $q_{X,Y} \triangleq \Lambda(F(\mathrm{app}) \circ p^{-1}_{Y^X,X}) : F(Y^X) \to (FY)^{(FX)}$, for $X, Y \in \mathrm{Obj}(\mathbb{C})$, have inverses. The cartesian closed endofunctors of $\mathbb{C}$, with natural transformations as morphisms, form a category, $\mathbf{End}^{\mathrm{cc}}_{\mathbb{C}}$. Like $\mathbf{End}^{\mathrm{s}}_{\mathbb{C}}$, $\mathbf{End}^{\mathrm{cc}}_{\mathbb{C}}$ is also a strict monoidal category with the monoidal product and the identity object defined in the same way. We shall use $\mathbf{End}^{\mathrm{cc}}_{\mathbb{C}}$ to build models for $\lambda^\circ$.

Let $\mathbb{C}$ be any bicartesian closed category. Let $\mathbf{S}$ be a strong monoidal functor from $\mathbf{C}(\mathcal{N})$ to $\mathbf{End}^{\mathrm{cc}}_{\mathbb{C}}$. Then, the interpretation, $[\![\_]\!]$, or more precisely $[\![\_]\!]_{(\mathbb{C},\mathbf{S})}$, of types and terms is as follows. The modal operator and contexts are interpreted as:

$$[\![\bigcirc A]\!] = \mathbf{S}_{0'}[\![A]\!] \qquad [\![\varnothing]\!] = \top \qquad [\![\Gamma, x :^n A]\!] = [\![\Gamma]\!] \times \mathbf{S}_n[\![A]\!]$$

Terms are interpreted as:

$$\begin{aligned}
[\![x]\!] &= [\![\Gamma_1]\!] \times \mathbf{S}_n[\![A]\!] \times [\![\Gamma_2]\!] \xrightarrow{\pi_i} \mathbf{S}_n[\![A]\!] \\
[\![\lambda x : A.b]\!] &= [\![\Gamma]\!] \xrightarrow{\Lambda[\![b]\!]} (\mathbf{S}_n[\![B]\!])^{(\mathbf{S}_n[\![A]\!])} \xrightarrow{q^{-1}} \mathbf{S}_n([\![B]\!]^{[\![A]\!]}) \\
[\![b\ a]\!] &= [\![\Gamma]\!] \xrightarrow{\langle q \circ [\![b]\!], [\![a]\!]\rangle} (\mathbf{S}_n[\![B]\!])^{(\mathbf{S}_n[\![A]\!])} \times \mathbf{S}_n[\![A]\!] \xrightarrow{\mathrm{app}} \mathbf{S}_n[\![B]\!] \\
[\![\mathbf{next}\ a]\!] &= [\![\Gamma]\!] \xrightarrow{[\![a]\!]} \mathbf{S}_{n+0'}[\![A]\!] \xrightarrow{\delta^{n,0'}} \mathbf{S}_n\mathbf{S}_{0'}[\![A]\!] \\
[\![\mathbf{prev}\ a]\!] &= [\![\Gamma]\!] \xrightarrow{[\![a]\!]} \mathbf{S}_n\mathbf{S}_{0'}[\![A]\!] \xrightarrow{\mu^{n,0'}} \mathbf{S}_{n+0'}[\![A]\!] \\
[\![\mathbf{inj}_1\ a_1]\!] &= [\![\Gamma]\!] \xrightarrow{[\![a_1]\!]} \mathbf{S}_n[\![A_1]\!] \xrightarrow{\mathbf{S}_n i_1} \mathbf{S}_n([\![A_1]\!] + [\![A_2]\!]) \\
[\![\mathbf{case}\ a\ \mathbf{of}\ b_1\ ;\ b_2]\!] &= [\![\Gamma]\!] \xrightarrow{\langle\langle [\![b_1]\!], [\![b_2]\!]\rangle, [\![a]\!]\rangle} (\mathbf{S}_n[\![B]\!]^{[\![A_1]\!]} \times \mathbf{S}_n[\![B]\!]^{[\![A_2]\!]}) \times \mathbf{S}_n([\![A_1]\!] + [\![A_2]\!]) \\
&\xrightarrow{p^{-1} \times \mathrm{id}} \mathbf{S}_n([\![B]\!]^{[\![A_1]\!]} \times [\![B]\!]^{[\![A_2]\!]}) \times \mathbf{S}_n([\![A_1]\!] + [\![A_2]\!]) \\
&\xrightarrow{p^{-1}} \mathbf{S}_n(([\![B]\!]^{[\![A_1]\!]} \times [\![B]\!]^{[\![A_2]\!]}) \times ([\![A_1]\!] + [\![A_2]\!])) \\
&\cong \mathbf{S}_n([\![B]\!]^{[\![A_1]\!]+[\![A_2]\!]} \times ([\![A_1]\!] + [\![A_2]\!])) \xrightarrow{\mathbf{S}_n \mathrm{app}} \mathbf{S}_n[\![B]\!]
\end{aligned}$$

This gives us a sound interpretation of $\lambda^\circ$.

**Theorem 2.21** If $\Gamma \vdash a :^n A$ in $\lambda^\circ$, then $[\![a]\!] \in \mathrm{Hom}_{\mathbb{C}}([\![\Gamma]\!], \mathbf{S}_n[\![A]\!])$. Further, if $\Gamma \vdash a_1 :^n A$ and $\Gamma \vdash a_2 :^n A$ such that $a_1 \equiv a_2$ in $\lambda^\circ$, then $[\![a_1]\!] = [\![a_2]\!] \in \mathrm{Hom}_{\mathbb{C}}([\![\Gamma]\!], \mathbf{S}_n[\![A]\!])$.

Such an interpretation is also computationally adequate, provided it is injective for ground types. The operational semantics for the calculus is assumed to be the call-by-value semantics induced by the $\beta$-rules in Figure 2.16. We denote the multi-step reduction relation corresponding to this operational semantics by $\longmapsto^*$. Formally, we can state adequacy as:

**Theorem 2.22** Let $\Gamma \vdash b :^0 \mathbf{Bool}$ and $v$ be a value of type **Bool**. If $[\![b]\!] = [\![v]\!]$ then $\vdash b \longmapsto^* v$.

Next, we use these categorical models to show correctness of binding-time analysis in $\lambda^\circ$.

### 2.9.3 Correctness of Binding-Time Analysis in $\lambda^\circ$

A binding-time analysis is correct if computation from an earlier stage *does not depend* upon computation from a later stage. Here, using the categorical model, we shall show that $\lambda^\circ$ satisfies this condition. We shall use the same *presence-absence test* technique that we used to prove noninterference for $\mathrm{DCC}_e$. The goal is to show that computations at a given stage can proceed when computations from all later stages are blacked out. Towards this end, we present a strong monoidal functor $\mathbf{S}^0$ that keeps untouched all computations at time instant 0 but blacks out all computations from every time instant greater than 0.

$$\mathbf{S}^0(n) = \begin{cases} \mathbf{Id} & \text{if } n = 0 \\ * & \text{otherwise} \end{cases}$$

Note that $\mathbf{S}^0$ is a strict monoidal functor with $\eta = \epsilon = \mathrm{id}$ and $\delta = \mu = \mathrm{id}$. Now, for any bicartesian closed category $\mathbb{C}$, any strong monoidal functor from $\mathbf{C}(\mathcal{N})$ to $\mathbf{End}^{\mathrm{cc}}_{\mathbb{C}}$ provides a computationally adequate interpretation of $\lambda^\circ$, given the interpretation for ground types is injective. Therefore,

**Theorem 2.23** $[\![\_]\!]_{(\mathbf{Set},\mathbf{S}^0)}$ is a computationally adequate interpretation of $\lambda^\circ$.

The existence of $\mathbf{S}^0$ shows that binding-time analysis in $\lambda^\circ$ is correct. We elaborate on this point below. By the above theorem, computations at time instant 0 can proceed independently of computations from all later stages. Once the computations from time instant 0 are done, we can move on to computations from time instant $0'$, which now acts like the new 0. Then, using the same argument, we can show that computations at time instant $0'$ can proceed independently of all later stage computations. Repeating this argument over and over again, we see that we can normalize $\lambda^\circ$-expressions in a time-ordered manner. Therefore, binding-time analysis in $\lambda^\circ$ is correct.

We can formalize the correctness argument into the following noninterference theorem.

**Theorem 2.24** Let $\varnothing \vdash f :^0 \bigcirc A \to \mathbf{Bool}$ and $\varnothing \vdash b_1 :^0 \bigcirc A$ and $\varnothing \vdash b_2 :^0 \bigcirc A$. Then, $\vdash f\ b_1 \longmapsto^* v$ if and only if $\vdash f\ b_2 \longmapsto^* v$, where $v$ is a value of type **Bool**.

The presence-absence test provides a simple yet powerful method for proving correctness of dependency analyses. We used it to show correctness of $\mathrm{DCC}_e$ and $\lambda^\circ$ in a very straightforward

manner. To put it in perspective, Davies [2017] requires 10 journal pages to establish correctness of $\lambda^\circ$ using syntactic methods whereas our proof of correctness for $\lambda^\circ$ follows almost immediately from the soundness theorem. This shows that the presence-absence test may be a useful tool for establishing correctness of dependency calculi.

Next, we show how to translate $\lambda^\circ$ into a graded monadic comonadic framework.

### 2.9.4 Can We Translate $\lambda^\circ$ to GMCC?

Here, we consider how we might translate $\lambda^\circ$ to GMCC. We can instantiate the parametrizing preordered monoid $\mathcal{M}$ to $\mathcal{N} = (\mathbb{N}, +, 0)$ (with discrete ordering) and we may translate $\bigcirc$ as: $\widehat{\bigcirc A} = S_{0'}\ \widehat{A}$. We can translate contexts as:

$$\widehat{\varnothing} = \varnothing \qquad \widehat{\Gamma, x :^n A} = \widehat{\Gamma}, x : S_n\ \widehat{A}$$

A typing judgment $\Gamma \vdash a :^n A$ can then be translated as: $\widehat{\Gamma} \vdash \widehat{a} : S_n\ \widehat{A}$. For the modal terms, we have:

$$\widehat{\mathbf{next}\ a} = \mathbf{fork}^{n,0'}\widehat{a} \qquad \widehat{\mathbf{prev}\ a} = \mathbf{join}^{n,0'}\widehat{a}$$

This translation works well for the modal constructs; however, we run into a problem when dealing with functions and applications. The problem is that with the above translation, it is not possible to show typing is preserved in case of functions and applications. The reason behind this problem is the difference between the calculi $\lambda^\circ$ and GMCC we referred to earlier: the types $\bigcirc^n(A \to B)$ and $\bigcirc^n A \to \bigcirc^n B$ are isomorphic in $\lambda^\circ$ but not necessarily in GMCC.

This difference arises from the fact that in $\lambda^\circ$, grades pervade all the typing rules, including the ones for functions and applications, while in GMCC, they are restricted to the monadic and comonadic typing rules. We could have designed GMCC by permeating the grades along all the typing rules in lieu of restricting them to a fragment of the calculus. We avoided such a design for the sake of simplicity. However, now that we understand the calculus, we can consider the implications of such a design choice. In the next section, we explore this design choice and present $\text{GMCC}_\text{e}$, GMCC extended with graded contexts and graded typing judgments.

## 2.10 $\text{GMCC}_\text{e}$

### 2.10.1 The Calculus

Like GMCC, $\text{GMCC}_\text{e}$ is parametrized by an arbitrary preordered monoid, $\mathcal{M}$. The types of the calculus are the same as those of GMCC. With respect to terms, $\text{GMCC}_\text{e}$ differs from GMCC in having only the following two term-level constructs (in lieu of **ret**, **extr**, etc.) for introducing

$\boxed{\Gamma \vdash a :^m A}$ *(Typing)*

$$\textsc{E-Var}\ \frac{}{\Gamma_1, x :^m A, \Gamma_2 \vdash x :^m A} \qquad \textsc{E-Lam}\ \frac{\Gamma, x :^m A \vdash b :^m B}{\Gamma \vdash \lambda x : A.b :^m A \to B} \qquad \textsc{E-App}\ \frac{\Gamma \vdash b :^m A \to B \quad \Gamma \vdash a :^m A}{\Gamma \vdash b\ a :^m B}$$

$$\textsc{E-Pair}\ \frac{\Gamma \vdash a_1 :^m A_1 \quad \Gamma \vdash a_2 :^m A_2}{\Gamma \vdash (a_1, a_2) :^m A_1 \times A_2} \qquad \textsc{E-Proj}\ \frac{\Gamma \vdash a :^m A_1 \times A_2}{\Gamma \vdash \mathbf{proj}_i\ a :^m A_i} \qquad \textsc{E-Inj}\ \frac{\Gamma \vdash a_i :^m A_i}{\Gamma \vdash \mathbf{inj}_i\ a_i :^m A_1 + A_2}$$

$$\textsc{E-Case}\ \frac{\Gamma \vdash a :^m A_1 + A_2 \quad \Gamma \vdash b_1 :^m A_1 \to B \quad \Gamma \vdash b_2 :^m A_2 \to B}{\Gamma \vdash \mathbf{case}\ a\ \mathbf{of}\ b_1\ ;\ b_2 :^m B} \qquad \textsc{E-Split}\ \frac{\Gamma \vdash a :^{m_1 \cdot m_2} A}{\Gamma \vdash \mathbf{split}^{m_2}\, a :^{m_1} S_{m_2}\, A} \qquad \textsc{E-Merge}\ \frac{\Gamma \vdash a :^{m_1} S_{m_2}\, A}{\Gamma \vdash \mathbf{merge}^{m_2}\, a :^{m_1 \cdot m_2} A}$$

$$\textsc{E-Unit}\ \frac{}{\Gamma \vdash \mathbf{unit} :^m \mathbf{Unit}} \qquad \textsc{E-Up}\ \frac{\Gamma \vdash a :^{m_1} A \qquad m_1 \leq m_2}{\Gamma \vdash a :^{m_2} A}$$

FIGURE 2.17: Typing Rules of GMCC$_\mathrm{e}$

and eliminating the modal type.

$$\text{terms}, a, b ::= \ldots (\lambda\text{-calculus terms}) \mid \mathbf{split}^m\, a \mid \mathbf{merge}^m\, a$$

GMCC$_\mathrm{e}$ also differs from GMCC wrt contexts and typing judgments, both of which are graded in GMCC$_\mathrm{e}$, like in $\lambda^\circ$. The typing judgment of GMCC$_\mathrm{e}$, $\Gamma \vdash a :^m A$, intuitively means that $a$ can be observed at $m$, provided the variables in $\Gamma$ are observable at their respective grades. The typing rules of the calculus appear in Figure 2.17. The rules pertaining to standard $\lambda$-calculus terms are as expected. The rules E-SPLIT and E-MERGE introduce and eliminate the modal type. These rules are similar to rule C-FORK and rule M-JOIN respectively. The rule E-UP, akin to rules M-UP and C-UP, implicitly relaxes the grade at which a term is observed.

The equational theory of the calculus is induced by the $\beta\eta$-rules of standard $\lambda$-calculus along with the following $\beta\eta$-rule for modal terms.

$$\mathbf{merge}^m(\mathbf{split}^m\, a) \equiv a \qquad a \equiv \mathbf{split}^m(\mathbf{merge}^m\, a)$$

The graded presentation of the calculus leads to some interesting consequences. Unlike GMCC, GMCC$_\mathrm{e}$ enjoys the following properties.

**Lemma 2.25** Let $\mathcal{M}$ be any preordered monoid. Then, in GMCC$_\mathrm{e}(\mathcal{M})$,

- The types $S_m\, \mathbf{Unit}$ and $\mathbf{Unit}$ are isomorphic.

- The types $S_m\ (A_1 \times A_2)$ and $S_m\ A_1 \times S_m\ A_2$, for all types $A_1$ and $A_2$, are isomorphic.
- The types $S_m\ (A \to B)$ and $S_m\ A \to S_m\ B$, for all types $A$ and $B$, are isomorphic.

The third property above reminds us of the isomorphism between types $\bigcirc^n(A \to B)$ and $\bigcirc^n A \to \bigcirc^n B$ in $\lambda^\circ$. Recall that we could not translate $\lambda^\circ$ into GMCC because such an isomorphism does not hold in general in GMCC. We shall see later, $\text{GMCC}_\text{e}$, that satisfies these isomorphisms, can readily capture $\lambda^\circ$.

Now, we build categorical models for $\text{GMCC}_\text{e}$. The models are similar to those of GMCC; however, as in the case of $\lambda^\circ$, we need to use cartesian closed endofunctors in lieu of just strong endofunctors. Let $\mathcal{M}$ be the parametrizing structure. Let $\mathbb{C}$ be any bicartesian closed category. Next, let $\mathbf{S}$ be a strong monoidal functor from $\mathbf{C}(\mathcal{M})$ to $\mathbf{End}^{\text{cc}}_{\mathbb{C}}$. Then, the interpretation $[\![\_]\!]$ of types, contexts and terms is as follows.

$$[\![S_m\ A]\!] = \mathbf{S}_m[\![A]\!] \qquad [\![\varnothing]\!] = \top \qquad [\![\Gamma, x :^m A]\!] = [\![\Gamma]\!] \times \mathbf{S}_m[\![A]\!]$$

$$[\![\mathbf{split}^{m_2}\, a]\!] = [\![\Gamma]\!] \xrightarrow{[\![a]\!]} \mathbf{S}_{m_1 \cdot m_2}[\![A]\!] \xrightarrow{\delta^{m_1,m_2}_{[\![A]\!]}} \mathbf{S}_{m_1}\mathbf{S}_{m_2}[\![A]\!]$$
$$[\![\mathbf{merge}^{m_2}\, a]\!] = [\![\Gamma]\!] \xrightarrow{[\![a]\!]} \mathbf{S}_{m_1}\mathbf{S}_{m_2}[\![A]\!] \xrightarrow{\mu^{m_1,m_2}_{[\![A]\!]}} \mathbf{S}_{m_1 \cdot m_2}[\![A]\!]$$

Note that the types and terms of the standard $\lambda$-calculus are interpreted as in the case of $\lambda^\circ$. This gives us a sound interpretation of the calculus.

**Theorem 2.26** If $\Gamma \vdash a :^m A$ in $\text{GMCC}_\text{e}$, then $[\![a]\!] \in \text{Hom}_{\mathbb{C}}([\![\Gamma]\!], \mathbf{S}_m[\![A]\!])$. Further, if $\Gamma \vdash a_1 :^m A$ and $\Gamma \vdash a_2 :^m A$ such that $a_1 \equiv a_2$ in $\text{GMCC}_\text{e}$, then $[\![a_1]\!] = [\![a_2]\!] \in \text{Hom}_{\mathbb{C}}([\![\Gamma]\!], \mathbf{S}_m[\![A]\!])$.

Next, we show that both $\text{DCC}_e$ and $\lambda^\circ$ can be translated to $\text{GMCC}_\text{e}$.

### 2.10.2 Translations from $\text{DCC}_e$ and $\lambda^\circ$ to $\text{GMCC}_\text{e}$

We know that, over the class of bounded join-semilattices, $\text{DCC}_e$ is equivalent to GMCC. Given that GMCC is quite close to $\text{GMCC}_\text{e}$, we shall use GMCC as the source language for our translation. Let $\mathcal{L} = (L, \sqcup, \bot)$ be an arbitrary join-semilattice. Then, the translation $\widetilde{\ }$ from $\text{GMCC}(\mathcal{L})$ to $\text{GMCC}_\text{e}(\mathcal{L})$ is as follows. For types, $\widetilde{A} = A$. For terms,

$$\widetilde{\mathbf{ret}\ a} = \mathbf{split}^{\bot}\widetilde{a} \qquad \widetilde{\mathbf{extr}\ a} = \mathbf{merge}^{\bot}\widetilde{a}$$
$$\widetilde{\mathbf{join}^{\ell_1,\ell_2}\, a} = \mathbf{split}^{\ell_1 \sqcup \ell_2}(\mathbf{merge}^{\ell_2}(\mathbf{merge}^{\ell_1}\widetilde{a})) \qquad \widetilde{\mathbf{fork}^{\ell_1,\ell_2}\, a} = \mathbf{split}^{\ell_1}(\mathbf{split}^{\ell_2}(\mathbf{merge}^{\ell_1 \sqcup \ell_2}\widetilde{a}))$$
$$\widetilde{\mathbf{lift}^{\ell} f} = \lambda x.\mathbf{split}^{\ell}(\widetilde{f}\ (\mathbf{merge}^{\ell} x)) \qquad \widetilde{\mathbf{up}^{\ell_1,\ell_2}\, a} = \mathbf{split}^{\ell_2}(\mathbf{merge}^{\ell_1}\widetilde{a})$$

The standard $\lambda$-calculus terms are translated as expected. This translation is sound, as we see next. Below, we use $\Gamma^m$ to denote the graded counterpart of $\Gamma$ where every assumption is held at grade $m$.

**Theorem 2.27** If $\Gamma \vdash a : A$ in GMCC($\mathcal{L}$), then $\Gamma^\perp \vdash \widetilde{a} :^\perp A$ in GMCC$_\mathrm{e}$($\mathcal{L}$). Further, if $\Gamma \vdash a_1 : A$ and $\Gamma \vdash a_2 : A$ such that $a_1 \equiv a_2$ in GMCC($\mathcal{L}$), then $\widetilde{a_1} \equiv \widetilde{a_2}$ in GMCC$_\mathrm{e}$($\mathcal{L}$).

One might wonder here whether we can go the other way around and translate GMCC$_\mathrm{e}$($\mathcal{L}$) to GMCC($\mathcal{L}$). Though the two calculi are very similar, such a translation is not possible, owing to Proposition 2.25. More concretely, note that the type $S_{\mathbf{High}}\,(S_{\mathbf{High}}\,\mathbf{Bool} \to \mathbf{Bool})$ has four distinct terms in GMCC$_\mathrm{e}$($\mathcal{L}_2$) but only two distinct terms in GMCC($\mathcal{L}_2$). Here, $\mathcal{L}_2 \triangleq \mathbf{Low} \sqsubset \mathbf{High}$.

Next, we return to our incomplete translation of $\lambda^\circ$ from Section 2.9.4. Though we couldn't translate $\lambda^\circ$ to GMCC($\mathcal{N}$), we can now easily translate it to GMCC$_\mathrm{e}$($\mathcal{N}$). The translation for the modal types and terms is as follows:

$$\widehat{\bigcirc A} = S_{0'}\,\widehat{A} \qquad \widehat{\mathbf{next}\ a} = \mathbf{split}^{0'}\widehat{a} \qquad \widehat{\mathbf{prev}\ a} = \mathbf{merge}^{0'}\widehat{a}$$

This translation is sound, as we see next. Below, $\widehat{\Gamma}$ denotes $\Gamma$ with the types of the assumptions translated.

**Theorem 2.28** If $\Gamma \vdash a :^n A$ in $\lambda^\circ$, then $\widehat{\Gamma} \vdash \widehat{a} :^n \widehat{A}$ in GMCC$_\mathrm{e}$($\mathcal{N}$). Further, if $\Gamma \vdash a_1 :^n A$ and $\Gamma \vdash a_2 :^n A$ such that $a_1 \equiv a_2$ in $\lambda^\circ$, then $\widehat{a_1} \equiv \widehat{a_2}$ in GMCC$_\mathrm{e}$($\mathcal{N}$).

GMCC$_\mathrm{e}$ is similar to the sealing calculus [Shikuma and Igarashi, 2006], which also subsumes the terminating fragment of DCC. Akin to **split** and **merge** in GMCC$_\mathrm{e}$, the sealing calculus uses **seal** and **unseal** to introduce and eliminate the modal type. However, the sealing calculus is less general than GMCC$_\mathrm{e}$ because it does not subsume $\lambda^\circ$. Additionally, the sealing calculus works for lattices only whereas GMCC$_\mathrm{e}$ works for any preordered monoid.

## 2.11 Discussions and Related Work

### 2.11.1 Nontermination

Till now, we did not include nontermination in any of our calculi. DCC, as presented by Abadi et al. [1999], includes nonterminating computations. So here we discuss how we can add nontermination to GMCC and GMCC$_\mathrm{e}$. One of the features of these calculi is that they can be decomposed into a standard $\lambda$-calculus fragment and a modal fragment. The categorical models for the calculi also reflect the separation between the two fragments. The $\lambda$-calculus fragment is modeled by a bicartesian closed category, $\mathbb{C}$, whereas the modal fragment is modeled using monoidal functors from the parametrizing monoid to the category of endofunctors over $\mathbb{C}$. This separation between the two fragments makes it easier to add nontermination to these calculi. To

include nontermination in these calculi, we just need to add the modal fragment on top of an already nonterminating calculus, for example, $\lambda$-calculus with pointed types. The nonterminating calculus can then be modeled by an appropriate category $\mathbb{D}$, for example, **Cpo**, and the modal fragment by monoidal functors from the parametrizing monoid to the category of endofunctors over $\mathbb{D}$. Owing to the separation between the fragments, we conjecture that our proofs, with straightforward modifications, can be carried over to the new setting.

### 2.11.2 Algehed's SDCC

Algehed [2018] is similar in spirit to our work. Algehed [2018] designs a calculus, SDCC, that is equivalent to (the terminating fragment of) DCC. SDCC has the same types as DCC. However, it replaces the **bind** construct of DCC with four new constructs: $\mu, map, up$ and $com$. The constructs $\mu, map$ and $up$ serve the same purpose as **join**, **lift** and **up** respectively. The construct $com$ is what separates SDCC from our work. While Algehed [2018] went with $com$ and designed a calculus equivalent to DCC, we went with **extr** and **fork** and designed a calculus that is equivalent to an extension of DCC. This choice helped us realize the power of comonadic aspect of dependency analysis.

### 2.11.3 Relational Semantics of DCC, Revisited

Abadi et al. [1999] present a relational categorical model for DCC. They interpret each of the $\mathcal{T}_\ell$s as a separate monad on a base category, $\mathcal{DC}$. However, their interpretation can also be phrased in terms of a strong monoidal functor from $\mathbf{C}(\mathcal{L})$ to $\mathbf{End}^s_{\mathcal{DC}}$. In other words, the model given by Abadi et al. [1999] is an instance of the general class of models admitted by $\mathrm{DCC}_e$, and hence DCC.

In this regard, we want to point out a technical problem with the category $\mathcal{DC}$. Since $\mathcal{DC}$ models simply-typed $\lambda$-calculus, it should be cartesian closed. Abadi et al. [1999] claim that it is so. However, contrary to their claim, the category $\mathcal{DC}$ is not cartesian closed. The exponential object in $\mathcal{DC}$ does not satisfy the universal property, unless the relations in the definition of $\mathrm{Obj}(\mathcal{DC})$ are restricted to reflexive ones only. Note that such modified objects are nothing but classified sets that Kavvos [2019] uses to build a categorical model of DCC based on cohesion. How the cohesion-based models relate to our graded models of DCC is something we would like to explore in future.

## 2.12 Conclusion

Dependency analysis is as much an analysis of dependence as of independence. By controlling the flow of information, it aims to ensure that certain entities are independent of certain other

entities while being dependent upon yet other entities. To control flow, it needs to make use of unidirectional devices, devices that allow flow along one direction but block along the other. The two simple yet robust unidirectional devices in programming languages are monads and comonads. Monads allow inflow but block outflow whereas comonads allow outflow but block inflow. When used together, they ensure that flow respects the constraints imposed upon it by a predetermined structure, like a preordered monoid. Such a controlled flow can be used for a variety of purposes like enforcing security constraints, analyzing binding-time, etc.

To wind up, this chapter goes through the nuances of dependency analysis in simple type systems. It establishes a foundation for the next chapter where we extend dependency analysis from simple to pure type systems.

# Chapter 3

# Dependency Analysis in Pure Type Systems

Even though many simple/polymorphic type systems have included dependency analysis in some form, this feature has seen relatively little attention in the context of dependent type systems. This is unfortunate because, as we show in this chapter, dependency analysis can provide an elegant foundation for compile-time and run-time irrelevance, two important concerns in the design of dependently-typed languages. Recall that compile-time irrelevance identifies sub-expressions that are not needed for type-checking while run-time irrelevance identifies sub-expressions that do not affect the result of evaluation. By ignoring or erasing such sub-expressions, compilers for dependently-typed languages increase the expressiveness of the type system, improve on compilation time and produce more efficient executables.

In this chapter, we augment a pure type system with a *primitive* notion of dependency analysis and use it to track compile-time and run-time irrelevance. We call this language DDC, for Dependent Dependency Calculus, in homage to DCC. Although our dependency analyses are structured differently, we show that DDC can faithfully embed the terminating fragment of DCC and support its many well-known applications, in addition to our novel applications of tracking compile-time and run-time irrelevance.

In this chapter, we make the following contributions:

- We design a language SDC, for Simple Dependency Calculus, that can analyze dependencies in a simply-typed language. SDC is equivalent to $\text{GMCC}_{\text{e}}$, when parametrized over bounded join-semilattices. Still, we present an alternative proof that shows SDC is no less expressive than the terminating fragment of DCC. The structure of dependency analysis in SDC enables a relatively straightforward syntactic proof of noninterference.

- We extend SDC to a dependent calculus, $\text{DDC}^{\top}$. Using this calculus, we analyze run-time irrelevance and show the analysis is correct using the noninterference theorem. $\text{DDC}^{\top}$ contains SDC as a sub-language. As such, it can be used to track other forms of dependencies as well.
- We generalize $\text{DDC}^{\top}$ to DDC. Using this calculus, we analyze both run-time and compile-time irrelevance and show that the analyses are correct. To the best of our knowledge, DDC is the only system that can distinguish run-time and compile-time irrelevance as separate modalities, necessary for the proper treatment of strong Σ-types with irrelevant first projections.

## 3.1 Irrelevance Analysis as a Dependency Analysis

### 3.1.1 Run-time Irrelevance

Parts of a program that are not required during run time are said to be run-time irrelevant. Our goal is to identify such parts. Let's consider some examples. We shall mark variables and arguments with $\top$ if they can be erased prior to execution and leave them unmarked if they should be preserved.

For example, the polymorphic identity function can be marked as:

```
id : Π x:⊤Type. x → x
id = λ⊤x. λy. y
```

The first parameter, $x$, of the identity function is only needed during type checking; it can be erased before execution. The second parameter, $y$, though, is required during runtime. When we apply this function to arguments, as in `(id Bool⊤ True)`, we can erase the first argument, `Bool`, but the second one, `True`, must be retained.

Indexed data structures provide another example of run-time irrelevance.

Consider the `Vec` datatype for length-indexed vectors, as it might look in a core language inspired by GHC [Sulzmann et al., 2007, Weirich et al., 2017]. The `Vec` datatype has two parameters, `n` and `a`, that also appear in the types of the data constructors, `Nil` and `Cons`. These parameters are relevant to `Vec`, but irrelevant to the data constructors. (Note that in the types of the constructors, the equality constraints `(n ~ Zero)` and `(n ~ Succ m)` force `n` to be equal to the length of the vector. Going forward, we shall elide these constraints because they are not central to our discussion on irrelevance.)

```
Vec  : Nat → Type → Type
Nil  : Π n:⊤Nat. Π a:⊤Type. (n ~ Zero) ⇒ Vec n a
Cons : Π n:⊤Nat. Π a:⊤Type. Π m:⊤Nat. (n ~ Succ m) ⇒ a → Vec m a → Vec n a
```

Now consider a function `vmap` that maps a given function over a given vector. The length of the vector and the type arguments are not necessary for running `vmap`; they are all erasable. So we assign them $\top$.

```
vmap : Π n:⊤Nat.Π a b:⊤Type.  (a → b) → Vec n a → Vec n b
vmap = λ⊤ n a b. λ f xs.
          case xs of
            Nil → Nil
            Cons m⊤ x xs → Cons m⊤ (f x) (vmap m⊤ a⊤ b⊤ f xs)
```

Note that the $\top$-marked variables `m`, `a` and `b` appear in the definition of `vmap`, but only in $\top$ contexts. By requiring that 'unmarked' terms *don't depend* on terms marked with $\top$, we can track run-time irrelevance and guarantee safe erasure. Observe that even though these arguments are marked with $\top$ to describe their use in the *definition* of `vmap`, this marking does not reflect their usage in the *type* of `vmap`. In particular, we are free to use these variables with `Vec` in a relevant manner.

### 3.1.2 Compile-time Irrelevance

Some type constructors may have arguments which can be ignored during type-checking. Such arguments are said to be *compile-time irrelevant*. For example, suppose we have a constant function that ignores its argument and returns a type.

```
phantom : Nat⊤ → Type
phantom = λ⊤ x.  Bool
```

To type-check `idp` below, we must show that `phantom 0` equals `phantom 1`. If we don't take compile-time irrelevance into account, we need to $\beta$-reduce both sides to show that the input and output types are equal. At times, such $\beta$-reductions may be costly, as we saw in an example in Section 1.2.5.

```
idp : phantom 0⊤ → phantom 1⊤
idp = λ x. x
```

If we take compile-time irrelevance into account, we can also use the dependency information contained in the type of a function to reason about it abstractly. For example, since the function `f` below ignores its argument, it is sound to equate the input and output types.

```
ida : Π f :⊤(Nat⊤ → Type).  f 0⊤ → f 1⊤
ida = λ⊤ f. λ x. x
```

In the absence of compile-time irrelevance analysis, we cannot type-check `ida`. So compile-time irrelevance analysis makes type-checking more flexible.

Analyzing compile-time irrelevance can also make type-checking faster when the types contain expensive computation that may be safely ignored. For example, consider the following program that type-checks without compile-time irrelevance analysis. However, in that case, the type-checker must show that `fib 28` reduces to `317811`, with `fib` being the Fibonacci function.

```
idn : Π f :⊤(Nat⊤ → Type). f (fib 28)⊤ → f 317811⊤
idn = λ⊤ f. λ x. x
```

So far, we have used two annotations on variables and terms: $\top$ for irrelevant ones and 'unmarked' for relevant ones. We used $\top$ to mark both arguments that can be erased at run time and arguments that can be safely ignored by the type-checker. However, sometimes we need a finer distinction.

### 3.1.3 Strong Irrelevant Σ-types

Consider the type $\Sigma$`m:`$^{\top}$`Nat. Vec m a`, which contains pairs whose first component is marked as irrelevant. This type might be useful, say, for the output of a `filter` function for vectors, where the length of the output vector cannot be calculated statically. If we never need to use this length at runtime, then it would be good to mark it with $\top$ so that it is not stored. However, note that it is safe for `m` to be used in a relevant position in the body of the Σ-type even when it is marked with $\top$. This marking indicates how the first component of a pair having this type is used, not how the bound variable `m` is used in the body of the type.

Now, marking `m` with $\top$ means that the first component of a pair having this type must also be *compile-time* irrelevant. This results in a significant limitation for strong Σ types: we cannot project the second component from the pair. To elaborate, say we have `ys:`$\Sigma$`m:`$^{\top}$`Nat. Vec m a`. The type of ($\pi_1$ `ys`) is a `Nat` that can only be used in irrelevant positions. However, the argument `n` in `Vec n a` must be compile-time relevant; otherwise the type-checker would equate `Vec 0 a` with `Vec 1 a`, making the length index meaningless. The type of ($\pi_2$ `ys`) would, then, be `Vec (`$\pi_1$ `ys) a`, which is ill-formed because an irrelevant term, `(`$\pi_1$ `ys)`, appears in a relevant position.

Therefore, we don't want to mark the first component of the output of `filter` with $\top$. However, if we leave it unmarked, we cannot erase it at runtime, something that we might want to. A way out of this quandry comes by considering terms that are run-time irrelevant but not compile-time irrelevant. Such terms exist between completely irrelevant and completely relevant terms. They should not depend upon irrelevant terms and relevant terms should not depend upon them. We mark such terms with a new annotation, $C$, with the constraints that 'unmarked' terms do not depend on $C$ and $C$ terms do not depend on $\top$ terms. The three annotations, then, correspond to the three levels of dependency lattice, with $\bot \sqsubset C \sqsubset \top$ (using $\bot$ in lieu of 'unmarked'). We call this lattice $\mathcal{L}_I$, for irrelevance lattice. Using this lattice, we can type-check the following `filter` function.

```
filter : Πn:⊤Nat.Πa:⊤Type.(a → Bool) → Vec n a → Σm:C Nat. Vec m a
filter = λ⊤ n a. λ f vec.
          case vec of
            Nil → (Zero^C, Nil)
            Cons m⊤ x xs
              | f x  → ((Succ (π1 ys))^C, Cons (π1 ys)⊤ x (π2 ys))
                           where
                             ys = filter m⊤ a⊤ f xs
              | _    → filter m⊤ a⊤ f xs
```

Eisenberg et al. [2021] observe that, in Haskell, it is important to use projection functions to access the components of the pair that results from the recursive call (as in $\pi_1$ `ys` and $\pi_2$ `ys`) to ensure that `filter` is not excessively strict. If `filter` instead used pattern matching to eliminate the pair returned by the recursive call, it would have needed to filter the entire vector before returning the first successful value. This `filter` function demonstrates the practical utility of strong irrelevant Σ-types because it supports the same run-time behavior of the usual list `filter` function but with more expressive data structures.

## 3.2 A Simple Dependency Analyzing Calculus

We start with an alternative calculus for dependency analysis in simple type systems, one that extends easily to dependent types. This calculus, SDC, for Simple Dependency Calculus, is parameterized by an arbitrary dependency lattice. An excerpt of this calculus appears in Figure 3.1; it is an extension of the simply-typed $\lambda$-calculus with a grade-indexed modal type $T^\ell\, A$. The calculus itself is also *graded*, which means that in a typing judgment, the derived term and every variable in the context carries a label or grade.

### 3.2.1 Type System

The typing judgment has the form $\Omega \vdash a :^\ell A$, which means that "$\ell$ is allowed to observe $a$" or that "$a$ is visible at $\ell$", assuming the variables in $\Omega$ are visible at their respective levels. Selected typing rules for SDC appear in Figure 3.1. Most rules are straightforward and propagate the level of the sub-terms to the expression.

The rule SDC-Var requires that the grade of the variable in the context must be less than or equal to the grade of the observer. In other words, an observer at level $\ell$ is allowed to use a variable from level $k$ if and only if $k \sqsubseteq \ell$. If the variable's level is too high, then this rule does not apply, ensuring that information flows to more secure levels but never to less secure ones. Abstraction rule SDC-Abs uses the current level of the expression for the newly introduced

*(Grammar)*

| | | | | |
|---|---|---|---|---|
| *labels* | $\ell, k$ | ::= | $\bot \mid \top \mid k \sqcap \ell \mid k \sqcup \ell \mid \ldots$ | |
| *types* | $A, B$ | ::= | $\mathbf{Unit} \mid A \to B \mid T^{\ell}\, A$ | |
| *terms* | $a, b$ | ::= | $x \mid \lambda x{:}A.a \mid a\ b$ | *variables and functions* |
| | | $\mid$ | $\eta^{\ell}\ a \mid \mathbf{bind}^{\ell}\ x = a\ \mathbf{in}\ b$ | *graded modality* |
| *contexts* | $\Omega$ | ::= | $\varnothing \mid \Omega, x{:}^{\ell} A$ | |

$\boxed{\Omega \vdash a :^{\ell} A}$ *(Typing rules)*

SDC-Var
$$\frac{\ell_0 \sqsubseteq \ell \qquad x{:}^{\ell_0} A \in \Omega}{\Omega \vdash x :^{\ell} A}$$

SDC-Abs
$$\frac{\Omega, x{:}^{\ell} A \vdash b :^{\ell} B}{\Omega \vdash \lambda x{:}A.b :^{\ell} A \to B}$$

SDC-App
$$\frac{\Omega \vdash b :^{\ell} A \to B \qquad \Omega \vdash a :^{\ell} A}{\Omega \vdash b\ a :^{\ell} B}$$

SDC-Return
$$\frac{\Omega \vdash a :^{\ell \sqcup \ell_0} A}{\Omega \vdash \eta^{\ell_0}\ a :^{\ell} T^{\ell_0} A}$$

SDC-Bind
$$\frac{\Omega \vdash a :^{\ell} T^{\ell_0} A \qquad \Omega, x{:}^{\ell \sqcup \ell_0} A \vdash b :^{\ell} B}{\Omega \vdash \mathbf{bind}^{\ell_0}\ x = a\ \mathbf{in}\ b :^{\ell} B}$$

$\boxed{a \rightsquigarrow a'}$ *(Small step)*

SDCStep-BindLeft
$$\frac{a \rightsquigarrow a'}{\mathbf{bind}^{\ell}\ x = a\ \mathbf{in}\ b \rightsquigarrow \mathbf{bind}^{\ell}\ x = a'\ \mathbf{in}\ b}$$

SDCStep-BindBeta
$$\frac{}{\mathbf{bind}^{\ell}\ x = \eta^{\ell}\ a\ \mathbf{in}\ b \rightsquigarrow b\{a/x\}}$$

Figure 3.1: Simple Dependency Calculus (Excerpt)

variable in the context. This makes sense because the argument to the function is checked at the same level in rule SDC-App.

The modal type, introduced and eliminated with rule SDC-Return and rule SDC-Bind respectively, manipulates the levels. Rule SDC-Return says that, if a term is $(\ell \sqcup \ell_0)$-secure, then we can put it in an $\ell_0$-secure box and release it at level $\ell$. An $\ell_0$-secure boxed term can be unboxed only by someone who has security clearance for $\ell_0$, as we see in rule SDC-Bind. The join operation in this rule ensures that $b$ can depend on $a$ only if $b$ itself is $\ell_0$-secure or $\ell_0 \sqsubseteq \ell$.

### 3.2.2 Meta-theoretic Properties

This type system satisfies several interesting properties with regard to levels.

First, we can always weaken our assumptions about the variables in the context. If a term is derivable with an assumption held at some grade, then that term is also derivable with that assumption held at any lower grade. Below, for any two contexts $\Omega_1, \Omega_2$, we say that $\Omega_1 \sqsubseteq \Omega_2$ iff they are the same modulo the grades and further, for any $x$, if $x{:}^{\ell_1} A \in \Omega_1$ and $x{:}^{\ell_2} A \in \Omega_2$, then $\ell_1 \sqsubseteq \ell_2$.

**Lemma 3.1** (Narrowing) If $\Omega' \vdash a :^{\ell} A$ and $\Omega \sqsubseteq \Omega'$, then $\Omega \vdash a :^{\ell} A$.

Narrowing says that we can always downgrade any variable in the context. Conversely, we cannot upgrade context variables in general, but we can upgrade them to the level of the judgment.

**Lemma 3.2** (Restricted Upgrading) If $\Omega_1, x :^{\ell_0} A, \Omega_2 \vdash b :^{\ell} B$ and $\ell_1 \sqsubseteq \ell$, then $\Omega_1, x :^{\ell_0 \sqcup \ell_1} A, \Omega_2 \vdash b :^{\ell} B$.

The restricted upgrading lemma is needed to show subsumption. Subsumption states that, if a term is visible at some grade, then it is also visible at all higher grades.

**Lemma 3.3** (Subsumption) If $\Omega \vdash a :^{\ell} A$ and $\ell \sqsubseteq k$, then $\Omega \vdash a :^{k} A$.

Subsumption is necessary (along with a standard weakening lemma) to show that substitution holds for this calculus. For substitution, we need to ensure that the level of the variable matches up with that of the substituted expression.

**Lemma 3.4** (Substitution) If $\Omega_1, x :^{\ell_0} A, \Omega_2 \vdash b :^{\ell} B$ and $\Omega_1 \vdash a :^{\ell_0} A$, then $\Omega_1, \Omega_2 \vdash b\{a/x\} :^{\ell} B$.

Now, let us look at the operational semantics of SDC. We use a call-by-name reduction strategy for SDC. An excerpt of the small-step reduction rules for SDC appears in Figure 3.1. Note how the grades on the introduction form and the corresponding elimination form match up in rule SDCStep-BindBeta. Further, note that we could have also used a call-by-value reduction strategy for SDC; we chose a call-by-name reduction strategy because our development is motivated by potential applications in Haskell.

For a call-by-name operational semantics, the above lemmas allow us to prove, a standard progress and preservation based type soundness result, which we omit here.

Note here that over the class of bounded join-semilattices, the calculi SDC and $\mathrm{GMCC}_{\mathrm{e}}$ are exactly equivalent. The reason we design SDC, then, is that it extends easily to dependent types and admits a relatively straightforward *syntactic* proof of noninterference. Next, we present this syntactic proof of noninterference for SDC.

### 3.2.3 A Syntactic Proof of Noninterference

When users with low-security clearance are oblivious to high-security data, we say that the system enjoys *noninterference*. Noninterference results from level-specific views of the world. The values $\eta^{\mathbf{High}}$ **True** and $\eta^{\mathbf{High}}$ **False** appear the same to a **Low**-user while a **High**-user can differentiate between them. To capture this notion of a level-specific view, we design an indexed equivalence relation on open terms, $\sim_\ell$, called *indexed indistinguishability*, and shown in Figure 3.2. To define this relation, we need the labels of the variables in the context but not their types. So, we use grade-only contexts $\Phi$, defined as $\Phi ::= \varnothing \mid \Phi, x : \ell$. These contexts are like graded contexts $\Omega$ without the type information on variables, also denoted by $|\Omega|$.

$\boxed{\Phi \vdash a \sim_\ell b}$ *(Indexed Indistinguishability)*

$$\textsc{Ind-Var}\ \frac{x : \ell_0\ \mathbf{in}\ \Phi \qquad \ell_0 \sqsubseteq \ell}{\Phi \vdash x \sim_\ell x} \qquad \textsc{Ind-Abs}\ \frac{\Phi, x\colon \ell \vdash b_1 \sim_\ell b_2}{\Phi \vdash \lambda x\colon A.b_1 \sim_\ell \lambda x\colon A.b_2} \qquad \textsc{Ind-App}\ \frac{\Phi \vdash b_1 \sim_\ell b_2 \quad \Phi \vdash a_1 \sim_\ell a_2}{\Phi \vdash b_1\ a_1 \sim_\ell b_2\ a_2}$$

$$\textsc{Ind-Return}\ \frac{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2}{\Phi \vdash \eta^{\ell_0}\ a_1 \sim_\ell \eta^{\ell_0}\ a_2} \qquad \textsc{Ind-Bind}\ \frac{\Phi \vdash a_1 \sim_\ell a_2 \quad \Phi, x\colon \ell_0 \sqcup \ell \vdash b_1 \sim_\ell b_2}{\Phi \vdash \mathbf{bind}^{\ell_0}\ x = a_1\ \mathbf{in}\ b_1 \sim_\ell \mathbf{bind}^{\ell_0}\ x = a_2\ \mathbf{in}\ b_2}$$

$\boxed{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2}$

$$\textsc{Eeq-Leq}\ \frac{\ell_0 \sqsubseteq \ell \qquad \Phi \vdash a_1 \sim_\ell a_2}{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2} \qquad \textsc{Eeq-Nleq}\ \frac{\neg(\ell_0 \sqsubseteq \ell)}{\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2}$$

FIGURE 3.2: Indexed indistiguishability for SDC (Excerpt)

Informally, $\Phi \vdash a \sim_\ell b$ means that $a$ and $b$ appear the same to an $\ell$-user. For example, $\eta^{\mathbf{High}}\ \mathbf{True} \sim_{\mathbf{Low}} \eta^{\mathbf{High}}\ \mathbf{False}$ but $\neg(\eta^{\mathbf{High}}\ \mathbf{True} \sim_{\mathbf{High}} \eta^{\mathbf{High}}\ \mathbf{False})$. We define this relation $\sim_\ell$ by structural induction on terms. We think of terms as ASTs annotated at various nodes with labels, say $\ell_0$, that determine whether an observer $\ell$ is allowed to look at the corresponding sub-tree. If $\ell_0 \sqsubseteq \ell$, then observer $\ell$ can start exploring the sub-tree; otherwise the entire sub-tree appears as a blob. So we can also read $\Phi \vdash a \sim_\ell b$ as: "$a$ is syntactically equal to $b$ at all parts of the terms marked with any label $\ell_0$, where $\ell_0 \sqsubseteq \ell$, but may be arbitrarily different elsewhere."

Note the rule IND-RETURN in Figure 3.2. It uses an auxiliary relation, $\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2$. This auxiliary *extended equivalence* relation $\Phi \vdash^{\ell_0}_{\ell} a_1 \sim a_2$ formalizes the idea discussed above: if $\ell_0 \sqsubseteq \ell$, then $a_1$ and $a_2$ must be indistinguishable at $\ell$; otherwise, they may be arbitrary terms.

Now, we explore some properties of the indistinguishability relation.If we remove the second component from an indistinguishability relation, $\Phi \vdash a \sim_\ell b$, we get a new judgment, $\Phi \vdash a : \ell$, called grading judgment. Now, corresponding to every indistinguishability rule, we define a grading rule where the indistinguishability judgments have been replaced with their grading counterparts. Terms derived using these grading rules are called well-graded. We can show that well-typed terms are well-graded.

**Lemma 3.5** (Typing implies grading) If $\Omega \vdash a :^{\ell} A$ then $|\Omega| \vdash a : \ell$.

The following lemma shows that indistinguishability is an equivalence relation. Observe that at the highest element of the lattice, $\top$, this equivalence degenerates to the identity relation.

**Lemma 3.6** (Equivalence) Indexed indistinguishability at $\ell$ is an equivalence relation on well-graded terms at $\ell$.

Indistinguishability is closed under extended equivalence, as we see below. The following is like a substitution lemma for the relation.

**Lemma 3.7** (Indistinguishability under substitution) If $\Phi, x : \ell \vdash b_1 \sim_k b_2$ and $\Phi \vdash^{\ell}_{k} a_1 \sim a_2$ then $\Phi \vdash b_1\{a_1/x\} \sim_k b_2\{a_2/x\}$.

With regard to the above lemma, consider the situation when $\neg(\ell \sqsubseteq k)$, for example, when $\ell = \mathbf{High}$ and $k = \mathbf{Low}$. In such a situation, for any two terms $a_1$ and $a_2$, if $\Phi, x : \ell \vdash b_1 \sim_k b_2$, then $\Phi \vdash b_1\{a_1/x\} \sim_k b_2\{a_2/x\}$. Let us work out a concrete example. For a typing derivation $x :^{\mathbf{High}} A \vdash b :^{\mathbf{Low}} \mathbf{Bool}$, we have, by lemmas 3.5 and 3.6, $x : \mathbf{High} \vdash b \sim_{\mathbf{Low}} b$. Then, $\varnothing \vdash b\{a_1/x\} \sim_{\mathbf{Low}} b\{a_2/x\}$. This is almost noninterference in action. What's left to show is that the indistinguishability relation respects the small step semantics:

**Theorem 3.8** (Noninterference) If $\Phi \vdash a_1 \sim_k a_1'$ and $a_1 \rightsquigarrow a_2$, then there exists some $a_2'$ such that $a_1' \rightsquigarrow a_2'$ and $\Phi \vdash a_2 \sim_k a_2'$.

Since the step relation is deterministic, in the above lemma, there is exactly one such $a_2'$ that $a_1'$ steps to. Now, going back to our last example, we see that $b\{a_1/x\}$ and $b\{a_2/x\}$ take steps in tandem and they are **Low**-indistinguishable after each and every step. Since the language itself is terminating, both the terms reduce to boolean values, values that are themselves **Low**-indistinguishable as well. But the indistinguishability relation for boolean values is just identity. This means that $b\{a_1/x\}$ and $b\{a_2/x\}$ reduce to the same value.

The indistinguishability relation gives us a syntactic method of proving noninterference for programs derived in SDC. Essentially, we show that a user with low-security clearance cannot distinguish between high security values just by observing program behavior.
Next, we show that SDC is no less expressive than the terminating fragment of DCC.

### 3.2.4 Relation with Sealing Calculus and Dependency Core Calculus

SDC is extremely similar to the sealing calculus $\lambda^{[]}$ of Shikuma and Igarashi [2006]. Like SDC, $\lambda^{[]}$ has a label on the typing judgment. But unlike SDC, $\lambda^{[]}$ uses standard ungraded typing contexts, $\Gamma$. Both the calculi have the same types. As far as terms are concerned, there is only one difference. The sealing calculus has an **unseal** term whereas SDC uses **bind**. We present the rules for sealing and unsealing terms in $\lambda^{[]}$ below. We take the liberty of making small cosmetic changes in the presentation.

$$\textsc{Sealing-Seal}\quad \frac{\Gamma \vdash a :^{\ell \sqcup \ell_0} A}{\Gamma \vdash \eta^{\ell_0}\ a :^{\ell} T^{\ell_0}\ A} \qquad \textsc{Sealing-Unseal}\quad \frac{\Gamma \vdash a :^{\ell} T^{\ell_0}\ A \qquad \ell_0 \sqsubseteq \ell}{\Gamma \vdash \mathbf{unseal}^{\ell_0}\, a :^{\ell} A}$$

Shikuma and Igarashi [2006] have shown that $\lambda^{[]}$ is equivalent to $\mathrm{DCC}_{\mathrm{pc}}$, an extension of the terminating fragment of DCC. Therefore, we compare SDC to DCC by simulating $\lambda^{[]}$ in SDC.

$$
\begin{array}{rcll}
a, A, b, B & ::= & s \mid \mathbf{unit} \mid \mathbf{Unit} & \textit{sorts and unit} \\
& \mid & \Pi x{:}^{\ell} A.B \mid x \mid \lambda x{:}^{\ell} A.a \mid a\ b^{\ell} & \textit{dependent functions} \\
& \mid & \Sigma x{:}^{\ell} A.B \mid (a^{\ell}, b) \mid \mathbf{let}\ (x^{\ell}, y)\ =\ a\ \mathbf{in}\ b & \textit{dependent pairs} \\
& \mid & A + B \mid \mathbf{inj}_1\ a \mid \mathbf{inj}_2\ a \mid \mathbf{case}\ a\ \mathbf{of}\ b_1; b_2 & \textit{disjoint unions}
\end{array}
$$

FIGURE 3.3: Dependent Dependency Calculus Grammar (Types and Terms)

For this, we define a translation $\overline{\cdot}$, from $\lambda^{[]}$ to SDC. Most of the cases are handled inductively in a straightforward manner. For **unseal**, we have, $\overline{\mathbf{unseal}^{\ell}\ a} := \mathbf{bind}^{\ell}\ x = \overline{a}\ \mathbf{in}\ x$.

With this translation, we can give a forward and a backward simulation connecting the two languages. The reduction relation $\rightsquigarrow$ below is full reduction for both the languages, the reduction strategy used by Shikuma and Igarashi [2006] for $\lambda^{[]}$. Full reduction is a non-deterministic reduction strategy whereby a $\beta$-redex in any sub-term may be reduced.

**Theorem 3.9** (Forward Simulation) If $a \rightsquigarrow a'$ in $\lambda^{[]}$, then $\overline{a} \rightsquigarrow \overline{a'}$ in SDC.

**Theorem 3.10** (Backward Simulation) For any term $a$ in $\lambda^{[]}$, if $\overline{a} \rightsquigarrow b$ in SDC, then there exists $a'$ in $\lambda^{[]}$ such that $b = \overline{a'}$ and $a \rightsquigarrow a'$.

The translation also preserves typing. In fact, a source term and its target have the same type. Below, for an ordinary context $\Gamma$, the graded context $\Gamma^{\ell}$ denotes $\Gamma$ with the labels for all the variables set to $\ell$.

**Theorem 3.11** (Translation Preserves Typing) If $\Gamma \vdash a :^{\ell} A$, then $\Gamma^{\ell} \vdash \overline{a} :^{\ell} A$.

The above translation shows that the terminating fragment of DCC can be embedded into SDC. Therefore, SDC is at least as expressive as the terminating fragment of DCC. Further, SDC lends itself nicely to syntactic proof technique for noninterference. This proof technique generalizes to more expressive systems, as we shall see in the next section, where we extend SDC to a general dependent dependency calculus.

## 3.3 A Dependent Dependency Analyzing Calculus

Here and in the next section, we present dependently-typed languages, with dependency analysis in the style of SDC. The first extension, called $\mathrm{DDC}^{\top}$, is a straightforward integration of labels and dependent types. $\mathrm{DDC}^{\top}$ subsumes SDC, and so can be used for the same purposes. In this section, we use $\mathrm{DDC}^{\top}$ to analyze *run-time irrelevance*. Then, in Section 3.4, we generalize $\mathrm{DDC}^{\top}$ to DDC that can also analyze *compile-time irrelevance*. We present these systems this way to emphasize that $\mathrm{DDC}^{\top}$ is an interesting calculus in its own right.

Both $\mathrm{DDC}^{\top}$ and DDC are pure type systems [Barendregt, 1993]. They share the same syntax, shown in Figure 3.3, combining terms and types into the same grammar. They are parameterized

$\boxed{\Omega \vdash a :^{\ell} A}$ *(Typing)*

$$\text{DCT-Var}\ \frac{\ell_0 \sqsubseteq \ell \qquad x :^{\ell_0} A \in \Omega}{\Omega \vdash x :^{\ell} A} \qquad \text{DCT-Type}\ \frac{\mathcal{A}(s_1, s_2)}{\Omega \vdash s_1 :^{\ell} s_2} \qquad \text{DCT-Pi}\ \frac{\Omega \vdash A :^{\ell} s_1 \qquad \Omega, x :^{\ell} A \vdash B :^{\ell} s_2 \qquad \mathcal{R}(s_1, s_2, s_3)}{\Omega \vdash \Pi x :^{\ell_0} A.B :^{\ell} s_3}$$

$$\text{DCT-Abs}\ \frac{\Omega, x :^{\ell_0 \sqcup \ell} A \vdash b :^{\ell} B \qquad \Omega \vdash (\Pi x :^{\ell_0} A.B) :^{\top} s}{\Omega \vdash \lambda x :^{\ell_0} A.b :^{\ell} \Pi x :^{\ell_0} A.B} \qquad \text{DCT-App}\ \frac{\Omega \vdash b :^{\ell} \Pi x :^{\ell_0} A.B \qquad \Omega \vdash a :^{\ell_0 \sqcup \ell} A}{\Omega \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}} \qquad \text{DCT-Conv}\ \frac{\Omega \vdash a :^{\ell} A \qquad |\Omega| \vdash A \equiv_{\top} B \qquad \Omega \vdash B :^{\top} s}{\Omega \vdash a :^{\ell} B}$$

FIGURE 3.4: $\text{DDC}^{\top}$ type system (core rules)

by a set of sorts $s$, a set of axioms $\mathcal{A}(s_1, s_2)$ which is a binary relation on sorts, and a set of rules $\mathcal{R}(s_1, s_2, s_3)$ which is a ternary relation on sorts.

We annotate several syntactic forms with grades for dependency analysis. The dependent function type, written $\Pi x :^{\ell} A.B$, includes the grade of the argument to a function having this type. Similarly, the dependent pair type, written $\Sigma x :^{\ell} A.B$, includes the grade of the first component of a pair having this type. We can interpret these types as a fusion of the usual, ungraded dependent types and the graded modality $T^{\ell}\, A$ we saw earlier. In other words, $\Pi x :^{\ell} A.B$ acts like the type $\Pi y : (T^{\ell}\, A).\mathbf{bind}^{\ell}\, x = y\, \mathbf{in}\, B$ and $\Sigma x :^{\ell} A.B$ acts like the type $\Sigma y : (T^{\ell}\, A).\mathbf{bind}^{\ell}\, x = y\, \mathbf{in}\, B$. Because of this fusion, we do not need to add the graded modality type as a separate form—we can define $T^{\ell}\, A$ as $\Sigma x :^{\ell} A.\mathbf{Unit}$. Using $\Pi x :^{\ell} A.B$ instead of $\Pi y : (T^{\ell}\, A).\mathbf{bind}\, x = y\, \mathbf{in}\, B$ has an advantage: the former allows $x$ to be held at differing grades while type checking $B$ and the body of a function having this $\Pi$-type while the latter requires $x$ to be held at the same grade in both the cases. We utilize this flexibility in Section 3.4.

### 3.3.1 $\text{DDC}^{\top}$ : $\Pi$-types

The core typing rules for $\text{DDC}^{\top}$ appear in Figure 3.4. As in the simple type system, the variables in the context are labeled and the judgment itself includes a label $\ell$. Rule DCT-Var is similar to its counterpart in the simply-typed language: the variable being observed must be graded less than or equal to the level of the observer. Rule DCT-Pi propagates the level of the expression to the sub-terms of the $\Pi$-type. Note that this type is annotated with an arbitrary label $\ell_0$: the purpose of this label $\ell_0$ is to denote the level at which the argument to a function having this type may be used.

In rule DCT-Abs, the parameter of the function is introduced into the context at level $\ell_0 \sqcup \ell$ (akin to rule SDC-Bind). In rule DCT-App, the argument to the function is checked at level $\ell_0 \sqcup \ell$ (akin to rule SDC-Return). Note that the $\Pi$-type is checked at $\top$ in rule DCT-Abs. In

$\mathrm{DDC}^{\top}$, level $\top$ corresponds to 'compile time' observers and motivates the superscript $\top$ in the language name.

Rule DCT-Conv converts the type of an expression to an equivalent type. The judgment $|\Omega| \vdash A \equiv_{\top} B$ is a label-indexed definitional equality relation instantiated to $\top$. This relation is the closure of the indexed indistinguishability relation (Section 3.2.3) under small-step call-by-name evaluation. When instantiated to $\top$, the relation degenerates to $\beta$-equivalence. So the rule DCT-Conv is essentially casting a term to a $\beta$-equivalent type; however, in the next section, we utilize the flexibility of label-indexing to cast a term to a type that may not be $\beta$-equivalent. Note here that similar to SDC, we use a call-by-name reduction strategy for $\mathrm{DDC}^{\top}$ and DDC. The small-step reduction relation, $\rightsquigarrow$, for these calculi are standard: however, the $\beta$-rules need to ensure that the grades on the introduction form and the corresponding elimination form match up, as in rule SDCStep-BindBeta, shown in Figure 3.1.

### 3.3.2 $\mathrm{DDC}^{\top}$ : $\Sigma$-types

The language $\mathrm{DDC}^{\top}$ includes $\Sigma$-types, as specified by the rules below.

$$\text{DCT-WSigma}\quad \frac{\Omega \vdash A :^{\ell} s_1 \qquad \Omega, x :^{\ell} A \vdash B :^{\ell} s_2 \qquad \mathcal{R}(s_1, s_2, s_3)}{\Omega \vdash \Sigma x{:}^{\ell_0} A.B :^{\ell} s_3}$$

$$\text{DCT-WPair}\quad \frac{\Omega \vdash a :^{\ell_0 \sqcup \ell} A \qquad \Omega \vdash b :^{\ell} B\{a/x\} \qquad \Omega \vdash \Sigma x{:}^{\ell_0} A.B :^{\top} s}{\Omega \vdash (a^{\ell_0}, b) :^{\ell} \Sigma x{:}^{\ell_0} A.B}$$

Like $\Pi$-types, $\Sigma$-types include a grade that is not related to how the bound variable is used in the body of the type. The grade indicates the level at which the first component of a pair having the $\Sigma$-type may be used. In rule DCT-WPair, we check the first component $a$ of the pair at a level raised by $\ell_0$, the level annotating the type, akin to rule SDC-Return. The second component $b$ is checked at the current level.

$$\text{DCT-LetPair}\quad \frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B \qquad \Omega, x{:}^{\ell_0 \sqcup \ell} A, y{:}^{\ell} B \vdash c :^{\ell} C\{(x^{\ell_0}, y)/z\} \qquad \Omega, z{:}^{\top} (\Sigma x{:}^{\ell_0} A.B) \vdash C :^{\top} s}{\Omega \vdash \mathbf{let}\ (x^{\ell_0}, y)\ =\ a\ \mathbf{in}\ c :^{\ell} C\{a/z\}}$$

The rule DCT-LetPair eliminates pairs using dependently-typed pattern matching. The pattern variables $x$ and $y$ are introduced into the context while checking the body $c$. Akin to rule SDC-Bind, the level of the first pattern variable, $x$, is raised by $\ell_0$. The result type $C$ is refined by the pattern-match, informing the type system that the pattern $(x^{\ell_0}, y)$ is equal to the scrutinee $a$.

Because of this refinement in the result type, we can define the projection operations through pattern-matching. In particular, the first projection, $\pi_1^{\ell_0}\ a \coloneqq \mathbf{let}\ (x^{\ell_0}, y)\ =\ a\ \mathbf{in}\ x$ while the second projection, $\pi_2^{\ell_0} a \coloneqq \mathbf{let}\ (x^{\ell_0}, y)\ =\ a\ \mathbf{in}\ y$. These projections can be type-checked according to the following derived rules:

$$\frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B \qquad \ell_0 \sqsubseteq \ell}{\Omega \vdash \pi_1^{\ell_0}\ a :^{\ell} A}\ \text{DCT-Proj1} \qquad \frac{\Omega \vdash a :^{\ell} \Sigma x{:}^{\ell_0} A.B}{\Omega \vdash \pi_2^{\ell_0} a :^{\ell} B\{\pi_1^{\ell_0}\ a/x\}}\ \text{DCT-Proj2}$$

Note that the derived rule DCT-Proj1 limits access to the first component through the premise $\ell_0 \sqsubseteq \ell$, akin to rule Sealing-Unseal. This condition makes sense because it aligns the observability of the first component of the pair with the label on the $\Sigma$-type.

### 3.3.3 Embedding SDC into DDC$^\top$

Here, we show how to embed SDC into DDC$^\top$.

We define a translation function, $\overline{\cdot}$, that takes the types and terms in SDC to terms in DDC$^\top$. For types, the translation is defined as: $\overline{\overline{A \to B}} \coloneqq {}^{\downarrow}\overline{\overline{A}} \to \overline{\overline{B}}$, $\overline{\overline{A \times B}} \coloneqq {}^{\downarrow}\overline{\overline{A}} \times \overline{\overline{B}}$ and $\overline{\overline{T^{\ell}\ A}} \coloneqq \Sigma x{:}^{\ell}\overline{\overline{A}}.\mathbf{Unit}$. (Here, ${}^{\ell}A \to B$ and ${}^{\ell}A \times B$ stand respectively for $\Pi x{:}^{\ell} A.B$ and $\Sigma x{:}^{\ell} A.B$, assuming $x$ is fresh.) For terms, the translation is straightforward except for the following cases: $\overline{\overline{\eta^{\ell}\ a}} \coloneqq (\overline{\overline{a}}^{\ell}, \mathbf{unit})$ and $\overline{\overline{\mathbf{bind}^{\ell}\ x = a\ \mathbf{in}\ b}} \coloneqq \mathbf{let}\ (x^{\ell}, y)\ =\ \overline{\overline{a}}\ \mathbf{in}\ \overline{\overline{b}}$, where $y$ is a fresh variable. By lifting the translation to contexts, we show it preserves typing.

**Theorem 3.12** (Trans. Preserves Typing) If $\Omega \vdash\ a :^{\ell}\ A$, then $\overline{\overline{\Omega}} \vdash \overline{\overline{a}} :^{\ell} \overline{\overline{A}}$.

Next, assuming a standard call-by-name small-step semantics for both the languages, we can provide a bisimulation.

**Theorem 3.13** (Forward Simulation) If $a \rightsquigarrow a'$ in SDC, then $\overline{\overline{a}} \rightsquigarrow \overline{\overline{a'}}$ in DDC$^\top$.

**Theorem 3.14** (Backward Simulation) For any term $a$ in SDC, if $\overline{\overline{a}} \rightsquigarrow b$ in DDC$^\top$, then there exists $a'$ in SDC such that $b = \overline{\overline{a'}}$ and $a \rightsquigarrow a'$.

Hence, SDC can be embedded into DDC$^\top$, preserving meaning. As such, DDC$^\top$ can analyze dependencies in general.

### 3.3.4 Run-time Irrelevance

Next, we show how to track run-time irrelevance using DDC$^\top$. We use the two element lattice $\{\bot, \top\}$ with $\bot \sqsubset \top$ such that $\bot$ and $\top$ correspond to run-time relevant and run-time irrelevant terms respectively. So, we need to erase terms marked with $\top$. However, we first define a general indexed erasure function, $\lfloor\cdot\rfloor_{\ell}$, on DDC$^\top$ terms, that erases everything an $\ell$-user should not be

able to see. The function is defined by straightforward recursion in most cases. For example, $\lfloor x \rfloor_\ell := x$ and $\lfloor \Pi x :^{\ell_0} A.B \rfloor_\ell := \Pi x :^{\ell_0} \lfloor A \rfloor_\ell . \lfloor B \rfloor_\ell$ and $\lfloor \lambda^{\ell_0} x.b \rfloor_\ell := \lambda^{\ell_0} x. \lfloor b \rfloor_\ell$.
The interesting cases are:
$\lfloor b\ a^{\ell_0} \rfloor_\ell := (\lfloor b \rfloor_\ell \lfloor a \rfloor_\ell^{\ell_0})$ if $\ell_0 \sqsubseteq \ell$ and $(\lfloor b \rfloor_\ell\ \mathbf{unit}^{\ell_0})$ otherwise,
$\lfloor (a^{\ell_0}, b) \rfloor_\ell := (\lfloor a \rfloor_\ell^{\ell_0}, \lfloor b \rfloor_\ell)$ if $\ell_0 \sqsubseteq \ell$ and $(\mathbf{unit}^{\ell_0}, \lfloor b \rfloor_\ell)$ otherwise.
They are so defined because if $\neg(\ell_0 \sqsubseteq \ell)$, an $\ell$-user should not be able to see $a$, so we replace it with **unit**.

This erasure function is closely related to the indistinguishability relation, we saw in Section 3.2.3, extended to a dependent setting. The erasure function maps the equivalence classes formed by the indistinguishability relation to their respective canonical elements.

**Lemma 3.15** (Canonical Element) If $\Phi \vdash a_1 \sim_\ell a_2$, then $\lfloor a_1 \rfloor_\ell = \lfloor a_2 \rfloor_\ell$.

Further, a well-graded term and its erasure are indistinguishable.

**Lemma 3.16** (Erasure Indistinguishability) If $\Phi \vdash a : \ell$, then $\Phi \vdash a \sim_\ell \lfloor a \rfloor_\ell$.

Next, we can show that erased terms simulate the reduction behavior of their unerased counterparts.

**Lemma 3.17** (Erasure Simulation) If $\Phi \vdash a : \ell$ and $a \rightsquigarrow b$, then $\lfloor a \rfloor_\ell \rightsquigarrow \lfloor b \rfloor_\ell$. Otherwise, if $a$ is a value, then so is $\lfloor a \rfloor_\ell$.

This lemma easily follows from Lemma 3.16 and the noninterference theorem (Theorem 3.8). Therefore, it is safe to erase, before run time, all sub-terms marked with $\top$.

This shows that we can correctly analyze run-time irrelevance using $\mathrm{DDC}^\top$. However, supporting compile-time irrelevance requires some changes to the system. We take them up in the next section.

## 3.4 DDC: Run-time and Compile-time Irrelevance

### 3.4.1 Towards Compile-time Irrelevance

Recall that terms which may be safely ignored while checking for type equality are said to be compile-time irrelevant. In $\mathrm{DDC}^\top$, the conversion rule DCT-Conv checks for type equality at $\top$.

$$\frac{\Omega \vdash a :^\ell A \qquad |\Omega| \vdash A \equiv_\top B \qquad \Omega \vdash B :^\top s}{\Omega \vdash a :^\ell B}\ \text{DCT-Conv}$$

The equality judgment used in this rule $\Phi \vdash a \equiv_\top b$ is an instantiation of the general judgment $\Phi \vdash a \equiv_\ell b$, which is the closure of the indistinguishability relation at $\ell$ under $\beta$-equivalence. When $\ell$ is $\top$, indistinguishability is just identity. As such, the equality relation at $\top$ degenerates

to standard $\beta$-equivalence. So, rule DCT-Conv does not ignore any part of the terms when checking for type equality.

To support compile-time irrelevance then, we need the conversion rule to use equality at some grade strictly less than $\top$ so that $\top$-marked terms may be ignored. For the irrelevance lattice $\mathcal{L}_I$, the level $C$ can be used for this purpose. For any other lattice $\mathcal{L}$, we can add two new elements, $C$ and $\top$, above every other existing element, such that $\mathcal{L} \sqsubset C \sqsubset \top$, and thereafter use level $C$ for this purpose. So, for any lattice, we can support compile-time irrelevance by equating types at $C$.

Referring back to the examples in Section 3.1.2, note that for `phantom : Nat`$^\top$ $\to$ `Type`, we have `phantom 0`$^\top \equiv_C$ `phantom 1`$^\top$. With this equality, we can type-check `idp = λ x. x : phantom 0`$^\top$ $\to$ `phantom 1`$^\top$, even without knowing the definition of `phantom`.

Now, observe that in rule DCT-Conv, the new type $B$ is also checked at $\top$. If we want to check for type equality at $C$, we need to make sure that the types themselves are checked at $C$. However, checking types at $C$ would rule out variables marked at $\top$ from appearing in them. This would restrict us from expressing many examples, including the polymorphic identity function.

To move out of this impasse, we take inspiration from EPTS [Mishra-Linger and Sheard, 2008, Mishra-Linger, 2008]. The key idea, adapted from Mishra-Linger and Sheard [2008], is to use a judgment of the form $C \sqcap \Omega \vdash a :^C A$ instead of a judgment of the form $\Omega \vdash a :^\top A$. The operation $C \sqcap \Omega$ takes the point-wise meet of the labels in the context $\Omega$ with $C$, essentially reducing any label marked as $\top$ to $C$, making it available for use in a $C$-expression. This operation, called *truncation*, makes $\top$ marked variables available at $C$. Other systems also use similar mechanisms for tracking irrelevance — for example, we can see a relation between this idea and analogous ones in Pfenning [2001] and Abel and Scherer [2012]. In these systems, "context resurrection" operation makes proof variables and irrelevant variables in the context available for use, similar to how $C \sqcap \Omega$ makes $\top$-marked variables in the context $\Omega$ available for use at $C$.

### 3.4.2 DDC: Basics

Next, we design a general dependency analyzing calculus, DDC, that takes advantage of compile-time irrelevance in its type system. DDC is a generalization of DDC$^\top$ and EPTS$^\bullet$ [Mishra-Linger and Sheard, 2008]. When $C$ equals $\top$, DDC degenerates to DDC$^\top$, that does not use compile-time irrelevance. When $C$ equals $\bot$, DDC degenerates to EPTS$^\bullet$, that identifies compile-time and run-time irrelevance. A crucial distinction between EPTS$^\bullet$ and DDC is that while the former is tied to a two element lattice, the latter can use any lattice. Thus, not only can DDC distinguish between run-time and compile-time irrelevance, but also it can simultaneously track other dependencies.

The core typing rules of DDC appear in Figure 3.5. Unlike DDC$^\top$, this type system maintains the invariant that $\ell \sqsubseteq C$ for all $\Omega \vdash a :^\ell A$. To ensure that this is the case, rule T-Type and

$\boxed{\Omega \vdash a :^{\ell} A}$ *(*DDC *core typing rules)*

$$\text{T-Var}\ \frac{x :^{\ell_0} A \in \Omega \qquad \ell_0 \sqsubseteq \ell \qquad \ell \sqsubseteq C}{\Omega \vdash x :^{\ell} A} \qquad \text{T-Type}\ \frac{\ell \sqsubseteq C \qquad \mathcal{A}(s_1, s_2)}{\Omega \vdash s_1 :^{\ell} s_2} \qquad \text{T-Pi}\ \frac{\Omega \vdash A :^{\ell} s_1 \qquad \Omega, x :^{\ell} A \vdash B :^{\ell} s_2 \qquad \mathcal{R}(s_1, s_2, s_3)}{\Omega \vdash \Pi x :^{\ell_0} A.B :^{\ell} s_3}$$

$$\text{T-AbsC}\ \frac{\Omega, x :^{\ell_0 \sqcup \ell} A \vdash b :^{\ell} B \qquad \Omega \Vdash (\Pi x :^{\ell_0} A.B) :^{\top} s}{\Omega \vdash \lambda x :^{\ell_0} A.b :^{\ell} \Pi x :^{\ell_0} A.B} \qquad \text{T-AppC}\ \frac{\Omega \vdash b :^{\ell} \Pi x :^{\ell_0} A.B \qquad \Omega \Vdash a :^{\ell_0 \sqcup \ell} A}{\Omega \vdash b\ a^{\ell_0} :^{\ell} B\{a/x\}} \qquad \text{T-ConvC}\ \frac{\Omega \vdash a :^{\ell} A \qquad |C \sqcap \Omega| \vdash A \equiv_C B \qquad \Omega \Vdash B :^{\top} s}{\Omega \vdash a :^{\ell} B}$$

$\boxed{\Omega \Vdash a :^{\ell} A}$ *(Truncate at* $\top$*)*

$$\text{CT-Leq}\ \frac{\Omega \vdash a :^{\ell} A \qquad \ell \sqsubseteq C}{\Omega \Vdash a :^{\ell} A} \qquad \text{CT-Top}\ \frac{C \sqcap \Omega \vdash a :^{C} A \qquad C \sqsubset \ell}{\Omega \Vdash a :^{\ell} A}$$

FIGURE 3.5: Dependent type system with compile-time irrelevance (core rules)

rule T-VAR include this precondition. This restriction means that we cannot really derive any term at $\top$ in DDC. We get around this restriction by deriving $C \sqcap \Omega \vdash a :^{C} A$ in lieu of $\Omega \vdash a :^{\top} A$.

Note that wherever DDC$^{\top}$ uses $\top$ as the observer level on a typing judgment, DDC uses truncation and level $C$ instead. If DDC$^{\top}$ uses some grade other than $\top$ as the observer level, DDC leaves the derivation as such. So a DDC$^{\top}$ judgment $\Omega \vdash a :^{\ell} A$ is replaced with a *truncated-at-top judgment*, $\Omega \Vdash a :^{\ell} A$ which can be read as: if $\ell = \top$, use the truncated version $C \sqcap \Omega \vdash a :^{C} A$; otherwise use the normal version $\Omega \vdash a :^{\ell} A$, as we see in Figure 3.5. In the typing rules, uses of this new judgment have been highlighted in gray to emphasize the modification with respect to DDC$^{\top}$.

### 3.4.3 Π-types

Rule T-PI is unchanged. The lambda rule T-ABSC now checks the type at $C$ after truncating the variables in the context to $C$. The application rule T-APPC checks the argument using the truncated-at-top judgment. Note that if $\ell_0 = \top$, the term $a$ can depend upon any variable in $\Omega$. Such a dependence is allowed since information can always flow from relevant to irrelevant contexts.

To see how irrelevance works in this system, let's consider the definition and use of the polymorphic identity function.

```
id : Π x:⊤Type. x → x
id = λ⊤x. λ y. y
```

In DDC$^\top$, the type `Π x:⊤Type. x → x` is checked at $\top$. However, here it must be checked at level $C$, which requires the premise, `x:`$^C$`Type ⊢ x → x :`$^C$ `Type`. Note that if we had used $\Pi y : T^\ell\, A.\mathbf{bind}^\ell\, x = y\, \mathbf{in}\, B$ to represent $\Pi x :^\ell A.B$, we would have been in trouble because `y :`$^C$ $T^\top$ `Type` $\not\vdash$ $\mathbf{bind}^\top$ `x = y in x → x :`$^C$ `Type`. This is why we fuse the graded modality with the dependent types.

Finally, observe that rule T-ConvC uses the definitional equality at $C$ instead of $\top$ and that the new type is checked after truncating the context. This rule shows how DDC supports compile-time irrelevance in its type system.

### 3.4.4 Σ-types

T-WPairC

$$\frac{\Omega \Vdash a :^{\ell_0 \sqcup \ell} A \qquad \Omega \vdash b :^\ell B\{a/x\} \qquad \Omega \Vdash \Sigma x{:}^{\ell_0} A.B :^\top s}{\Omega \vdash (a^{\ell_0}, b) :^\ell \Sigma x{:}^{\ell_0} A.B}$$

T-LetPairC

$$\frac{\Omega \vdash a :^\ell \Sigma x{:}^{\ell_0} A.B \qquad \Omega, x{:}^{\ell_0 \sqcup \ell} A, y{:}^\ell B \vdash c :^\ell C\{(x^{\ell_0}, y)/z\} \qquad \Omega, z{:}^\top (\Sigma x{:}^{\ell_0} A.B) \Vdash C :^\top s}{\Omega \vdash \mathbf{let}\ (x^{\ell_0}, y)\ =\ a\ \mathbf{in}\ c :^\ell C\{a/z\}}$$

We also need to modify the typing rules for Σ-types accordingly. In particular, when we create a pair, we check the first component using the truncated-at-top judgment. This is akin to how we check the argument in rule T-AppC. The rule T-LetPairC is same as rule DCT-LetPair, other than the fact that it checks the return type after truncating the context.

Note that in rule T-WPairC, if $\ell_0 = \top$, the first component $a$ is compile-time irrelevant. In such a situation, we cannot use projections to eliminate the pair because we would not be able to type-check the second projection whose type depends upon the first projection. So the derived second-projection rule T-Proj2C, shown below, includes the side-condition $\ell_0 \sqsubseteq C$. Thus, pairs having type $\Sigma x{:}^\top A.B$ can only be eliminated via pattern-matching, provided $B$ mentions $x$. However, pairs having type $\Sigma x{:}^{\ell_0} A.B$, where $\ell_0 \sqsubset \top$, can be eliminated via projections. For example, consider the output of the `filter` function, `ys :Σm:`$^C$`Nat.Vec m a`. We can eliminate this output via projections: $\pi_1$ `ys :`$^C$ `Nat` and $\pi_2$ `ys : Vec (`$\pi_1$ `ys) a`. Note that ($\pi_1$ `ys`) is used in the type of ($\pi_2$ `ys`). We can substitute ($\pi_1$ `ys :`$^C$ `Nat`) for `m` in (`Vec m a`) because `a :`$^C$ `Type, m :`$^C$`Nat ⊢ Vec m a :`$^C$ `Type`. Further, note that ($\pi_1$ `ys`) cannot be used at $\bot$, where it may be safely erased.

T-Proj1C

$$\frac{\Omega \vdash a :^\ell \Sigma x{:}^{\ell_0} A.B \qquad \ell_0 \sqsubseteq \ell}{\Omega \vdash \pi_1^{\ell_0}\ a :^\ell A}$$

T-Proj2C

$$\frac{\Omega \vdash a :^\ell \Sigma x{:}^{\ell_0} A.B \qquad \ell_0 \sqsubseteq C}{\Omega \vdash \pi_2^{\ell_0} a :^\ell B\{\pi_1^{\ell_0}\ a/x\}}$$

### 3.4.5 Noninterference

DDC satisfies a noninterference theorem analogous to the one for SDC. Recall that the noninterference theorem for SDC is proved via the auxiliary relations of *grading*, $\Phi \vdash a : \ell$, and *indexed indistiguishability*, $\Phi \vdash b_1 \sim_\ell b_2$. These relations can be straightforwardly extended from SDC to DDC. With these relations extended to DDC, we can prove the noninterference theorem for the calculus in a similar manner. Below, we present some of the key lemmas needed for proving this theorem. Note that these lemmas are analogues of the ones in Section 3.2.3.

**Lemma 3.18** (Typing implies grading) If $\Omega \vdash a :^\ell A$ then $|\Omega| \vdash a : \ell$.

**Lemma 3.19** (Equivalence) Indexed indistinguishability at $\ell$ is an equivalence relation on well-graded terms at $\ell$.

**Lemma 3.20** (Indistinguishability under substitution) If $\Phi, x : \ell \vdash b_1 \sim_k b_2$ and $\Phi \vdash^\ell_k a_1 \sim a_2$ then $\Phi \vdash b_1\{a_1/x\} \sim_k b_2\{a_2/x\}$.

**Theorem 3.21** (Noninterference for DDC) If $\Phi \vdash a_1 \sim_k a_1'$ and $a_1 \rightsquigarrow a_2$ then there exists some $a_2'$ such that $a_1' \rightsquigarrow a_2'$ and $\Phi \vdash a_2 \sim_k a_2'$.

### 3.4.6 Consistency of Definitional Equality

To show that the type system of DDC is sound, we need to show that the definitional equality relation is consistent. Consistency of definitional equality means that there is no derivation that equates two types having different *head forms*. For example, it should not equate **Nat** with **Unit** or **Bool** with $\Pi x :^\ell A.B$. Note that head forms are syntactic forms that correspond to types, such as, sorts $s$, **Unit**, $\Pi x :^\ell A.B$, etc.

Now, the definitional equality relation of DDC incorporates compile-time irrelevance. As such, if $\top$-inputs interfere with $C$-outputs in the calculus, this relation cannot be consistent. To see why, suppose $\top$-inputs interfere with $C$-outputs: let $x :^\top A \vdash b :^C$ **Bool** and $a_1, a_2 : A$ such that the terms $b\{a_1/x\}$ and $b\{a_2/x\}$ reduce to **True** and **False** respectively. Now we have, $(\lambda^\top x.\mathbf{if}\ b\ \mathbf{then\ Nat\ else\ Unit})\ {a_1}^\top \equiv_C (\lambda^\top x.\mathbf{if}\ b\ \mathbf{then\ Nat\ else\ Unit})\ {a_2}^\top$. But then, by $\beta$-equivalence, **Nat** $\equiv_C$ **Unit**. Therefore, consistency of the definitional equality relation of DDC hinges upon noninterference.

Consistency of definitional equality in a dependent calculus is usually proved via parallel reduction [Weirich et al., 2017]. However, with regard to DDC, we need to modify the standard parallel-reduction based proofs to take noninterference into account. Below, we briefly describe the necessary modifications. First, we prove that if two terms are definitionally equal at $\ell$, then they are joinable at $\ell$, meaning they reduce, through parallel reduction, to two terms that are indistinguishable at $\ell$. Next, we show that joinability at $\ell$ implies consistency. We, then, conclude that for any $\ell$, the definitional equality relation at $\ell$ is consistent. Note how the joinability relation in DDC is different from a standard joinability relation [Weirich et al., 2017]: here, the relation is indexed and uses indexed indistinguishability in lieu of standard syntactic equality.

Since the definitional equality relation (at any $\ell$) is consistent, the conversion rule T-ConvC (which checks for definitional equality at $C$), is sound. Therefore, DDC tracks compile-time irrelevance correctly. (Note that DDC can track run-time irrelevance in the same way as DDC$^\top$.) Below, we formally state consistency in terms of head forms.

**Theorem 3.22** (Consistency) If $\Phi \vdash a \equiv_\ell b$, and both $a$ and $b$ are head forms, then they have the same head form.

### 3.4.7 Soundness Theorem

DDC is type-sound. Below, we give an overview of the important lemmas that help establish soundness.

The properties below are stated for DDC, but they also apply to DDC$^\top$ since DDC degenerates to DDC$^\top$ whenever $C = \top$. First, we list the properties related to grading that hold for all judgments: indexed indistinguishability, definitional equality, and typing. (We only state the lemmas for typing, their counterparts for indexed indistinguishability and definitional equality are analogous.) These lemmas are similar to their simply-typed counterparts in Section 3.2.2.

**Lemma 3.23** (Narrowing) If $\Omega \vdash a :^\ell A$ and $\Omega' \sqsubseteq \Omega$, then $\Omega' \vdash a :^\ell A$

**Lemma 3.24** (Weakening) If $\Omega_1, \Omega_2 \vdash a :^\ell A$, then $\Omega_1, \Omega, \Omega_2 \vdash a :^\ell A$.

**Lemma 3.25** (Restricted upgrading) If $\Omega_1, x :^{\ell_0} A, \Omega_2 \vdash b :^\ell B$ and $\ell_1 \sqsubseteq \ell$, then $\Omega_1, x :^{\ell_0 \sqcup \ell_1} A, \Omega_2 \vdash b :^\ell B$.

Next, we list some properties that are specific to the typing judgment. For any typing judgment in DDC, the observer grade is at most $C$. Further, the observer grade of any judgment can be raised up to $C$.

**Lemma 3.26** (Bounded by $C$) If $\Omega \vdash a :^\ell A$ then $\ell \sqsubseteq C$.

**Lemma 3.27** (Subsumption) If $\Omega \vdash a :^\ell A$ and $\ell \sqsubseteq k$ and $k \sqsubseteq C$ then $\Omega \vdash a :^k A$

Next, note that we don't require contexts to be well-formed in the typing judgments; we add context well-formedness constraints, as required, to our lemmas. The following lemmas are true for well-formed contexts. A context $\Omega$ is well-formed, expressed as $\vdash \Omega$, iff for any assumption $x :^\ell A$ in $\Omega$, we have, $\Omega' \Vdash A :^\top s$, where $\Omega'$ is the prefix of $\Omega$ that appears before the assumption $x :^\ell A$.

**Lemma 3.28** (Substitution) If $\Omega_1, x :^{\ell_0} A, \Omega_2 \vdash b :^\ell B$ and $\vdash \Omega_1$ and $\Omega_1 \Vdash a :^{\ell_0} A$, then $\Omega_1, \Omega_2\{a/x\} \vdash b\{a/x\} :^\ell B\{a/x\}$

Finally, we have the two main lemmas proving type soundness.

**Lemma 3.29** (Preservation) If $\Omega \vdash a :^\ell A$ and $\vdash \Omega$ and $a \rightsquigarrow a'$, then $\Omega \vdash a' :^\ell A$.

**Lemma 3.30** (Progress) If $\varnothing \vdash a :^\ell A$ then either $a$ is a value or there exists some $a'$ such that $a \rightsquigarrow a'$.

These lemmas show that DDC is type-sound. We have already seen that DDC analyzes run-time and compile-time irrelevance correctly.

Next, we consider type-checking in DDC. DDC is parameterized by a generic pure type system and a generic lattice. When the parameterizing pure type system is strongly normalizing, such as the Calculus of Constructions, type-checking in DDC is decidable. In the next section, we demonstrate decidability of type-checking.

## 3.5 Type-Checking in DDC

Being a generic Pure Type System, not all instances of DDC admit decidable type-checking. For example, in the presence of the `Type:Type` axiom, DDC includes non-terminating computations via Girard's paradox. As as a result, we cannot decide equality in this instance of DDC, so type-checking will be undecidable. However, if the sorts, axioms and rules are chosen such that the underlying Pure Type System is strongly normalizing, then we can define a decidable type-checking algorithm for DDC. The algorithm is standard, but relies on a decision procedure for the equality judgment.

Our consistency proof, described in Section 3.4.6, gives us a start in designing a decision procedure for the equality judgment. This proof uses an auxiliary binary relation called *joinability*, which holds when two terms can use multiple steps of parallel reduction to reach to two terms that differ only in their unobservable components. Joinability and definitional equality induce the same relation on DDC terms: we can show that two DDC terms are definitionally equal if and only if they are joinable. This means that a decision procedure based on joinability will be sound and complete for DDC's definition of equivalence. Therefore, decidability of type-checking in DDC reduces to showing strong normalization.

If we select the sorts, axioms and rules of DDC to match those of a strongly normalizing calculus, for example, the Calculus of Constructions [Barendregt, 1993], we believe that strong normalization holds for the calculus, but leave a direct proof for future work. Our belief is supported by the fact that a sublanguage of this instance of DDC is strongly normalizing. We show this fact by translating this sublanguage to ICC$^*$, a strongly normalizing calculus.

ICC$^*$ [Barras and Bernardo, 2008], is a version of the Implicit Calculus of Constructions [Miquel, 2001] with annotations that support decidable type checking. It includes relevant and irrelevant Π-types only. So for our translation, we restrict attention to the corresponding fragment of DDC. We define the following translation, written $\widetilde{\cdot}$, and shown in Figure 3.6, that converts DDC terms to ICC$^*$ terms. The key point to note is that this translation maps arguments labeled $C$ and below to relevant arguments, and those labeled greater than $C$, such as $\top$, to irrelevant arguments. Note here that the syntax of ICC$^*$ uses parentheses to indicate relevant arguments and square brackets to indicate arguments that are irrelevant.

$$\widetilde{x} = x \qquad \widetilde{s} = s \qquad \widetilde{\Pi x{:}^{\ell} A.B} = \begin{cases} \Pi(x{:}\widetilde{A}).\widetilde{B} & \text{if } \ell \sqsubseteq C \\ \Pi[x{:}\widetilde{A}].\widetilde{B} & \text{otherwise} \end{cases}$$

$$\widetilde{\lambda x{:}^{\ell} A.b} = \begin{cases} \lambda(x{:}\widetilde{A}).\widetilde{b} & \text{if } \ell \sqsubseteq C \\ \lambda[x{:}\widetilde{A}].\widetilde{b} & \text{otherwise} \end{cases} \qquad \widetilde{b\ a^{\ell}} = \begin{cases} \widetilde{b}\ (\widetilde{a}) & \text{if } \ell \sqsubseteq C \\ \widetilde{b}\ [\widetilde{a}] & \text{otherwise} \end{cases}$$

FIGURE 3.6: Translation from a sublanguage of DDC to ICC*

We show that the above translation preserves definitional equality and typing. Note that ICC* compares terms for equality after an erasure operation, written $\cdot^*$, that removes all irrelevant arguments. Here, $\widetilde{\Omega}$ denotes $\Omega$ with the labels at the variable bindings omitted and the types translated.

**Lemma 3.31** (Translation preservation) If $\Phi \vdash A \equiv_C B$, then $\widetilde{A}^* \cong_{\beta\eta} \widetilde{B}^*$.
If $\Omega \vdash a :^{\ell} A$, then $\widetilde{\Omega} \vdash \widetilde{a} : \widetilde{A}$.

Next, note that $\beta$-reductions are also preserved by this translation. Therefore, since ICC* is a strongly normalizing calculus, we can conclude the same for this instance of DDC as well.

**Non-terminating instances of DDC.** For Pure Type Systems that are not strongly normalizing, such as the `Type:Type` language, there is an alternative approach to developing a calculus with decidable type-checking, following Weirich et al. [2017]. The key idea is to develop an annotated version of the calculus that book-keeps additional information from typing and equality derivations. In such an annotated version, the conversion rule would include an explicit coercion annotation that witnesses the equality between the concerned types, thus avoiding the need for normalization. Therefore, even for non-terminating instances of DDC, one can design a type-checking algorithm, after appropriately annotating the calculus.

## 3.6 Discussions and Related Work

### 3.6.1 Irrelevance in Dependent Type Theories

Analysis of compile-time and run-time irrelevance in dependent type theories is a well-studied topic [Abel and Scherer, 2012, Atkey, 2018, Barras and Bernardo, 2008, Brady, 2021, McBride, 2016, Miquel, 2001, Mishra-Linger and Sheard, 2008, Mishra-Linger, 2008, Moon et al., 2021, Nuyts and Devriese, 2018, Pfenning, 2001, Tejiščák, 2020, Weirich et al., 2017, etc]. To compare with literature, system DDC$^{\top}$, presented in this chapter, can support run-time irrelevance and is similar to the core language of Tejiščák [2020]. However, note that DDC$^{\top}$ can analyze dependencies in general while the system in Tejiščák [2020] is hardwired for run-time irrelevance analysis only.

On the other hand, DDC is the only system that we are aware of that analyzes run-time and compile-time irrelevance separately and makes use of the latter in the conversion rule. Further, DDC analyzes these irrelevances in the presence of strong Σ-types with erasable first components, something which, to the best of our knowledge, no prior work has been able to.

Prior work has identified the difficulty in handling strong Σ-types with erasable first components in a setting that analyzes compile-time irrelevance. Abel and Scherer [2012] point out that such Σ-types make their theory inconsistent. Similarly, EPTS$^\bullet$ [Mishra-Linger, 2008] cannot define the projections for pairs having such Σ-types. The reason behind this is that EPTS$^\bullet$ is hardwired to work with a two-element lattice that leads to an identification of compile-time and run-time irrelevance. As such, projections from such pairs result in type unsoundness. For example, considering the first components to be run-time irrelevant, the pairs $(\mathbf{Int}, \mathbf{unit})$ and $(\mathbf{Bool}, \mathbf{unit})$ are run-time equivalent. Since EPTS$^\bullet$ identifies run-time and compile-time irrelevance, these pairs are also compile-time equivalent. Then, taking the first projections of these pairs, one ends up with **Int** and **Bool** being compile-time equivalent, which is unsound. We resolve this problem by distinguishing between run-time and compile-time irrelevance, thus requiring a lattice with three elements.

Next, we compare our work with existing literature with respect to the equality relation. We analyze compile-time irrelevance to enable the equality relation to ignore unnecessary sub-terms. However, since our equality relation is untyped, we cannot include type-dependent rules in our system, such as $\eta$-equivalence for the **Unit** type. Several prior works on irrelevance [Barras and Bernardo, 2008, Miquel, 2001, Mishra-Linger, 2008, Tejiščák, 2020] use an untyped equality relation. However, some prior work, such as Abel and Scherer [2012], Pfenning [2001], do consider compile-time irrelevance in the context of type-directed equality. But such systems require irrelevant arguments to functions appear only irrelevantly in the codomain type of the function, thus ruling out several examples, including the polymorphic identity function.

### 3.6.2 Linear and BLL-Based Type Systems

The type systems presented in this chapter are closely related to linear and BLL-based type systems [Abel and Bernardy, 2020, Atkey, 2018, Brunel et al., 2014, Ghica and Smith, 2014, McBride, 2016, Moon et al., 2021, Orchard et al., 2019, Petricek et al., 2014]. Recall that such systems are parametrized by preordered semirings and provide a fine-grained accounting of resource usage. A typical judgment from a BLL-based type system looks like:

$$x :^1 \mathbf{Bool}, y :^1 \mathbf{Int}, z :^0 \mathbf{Bool} \vdash \mathbf{if}\, x\, \mathbf{then}\, y + 1\, \mathbf{else}\, y - 1 \;:^1\; \mathbf{Int}$$

The variable $x$ is used once in the condition, the variable $y$ is used once in each of the branches while the variable $z$ is not used at all. As such, they are marked with these grades in the context.

This form of judgment is very similar to the typing judgments of the systems presented in this chapter, with $q \in Q$ appearing in place of $\ell \in L$. However, there is a crucial difference: to the right of the turnstile, while any level, $\ell \in L$, may appear in the judgments of systems presented in this chapter, only the grade $1 \in Q$ can appear in typing judgments of BLL-based systems. A BLL-based system that allows an arbitrary grade to the right of the turnstile is not closed under substitution [Atkey, 2018, McBride, 2016]. As such, these systems are tied to a fixed reference while the systems in this chapter can vary their reference levels. This difference in form results from the difference in the purposes the two kinds of systems serve: BLL-based systems count while dependency systems compare. Counting does not require a variable reference whereas comparison does. As such, applications that require counting, like linearity tracking, are handled well by BLL-based systems while applications that require comparison, like ensuring secure information flow, are handled well by systems presented in this chapter. In Chapter 5, we shall see how these two kinds of systems can be brought together.

### 3.6.3 Dependency Analysis and Dependent Type Theory

Dependency analysis and dependent type theories have come together in some existing work.

Prost [2000] extends the $\lambda$-cube so that it may track dependencies, similar to what we do for Pure Type Systems. However, unlike our approach, Prost [2000] uses sorts to track dependencies. This approach is inspired by the distinction between different sorts in the Calculus of Constructions, where computationally relevant and irrelevant terms live in sorts `Set` and `Prop` respectively. However, such an approach ties up two distinct language features, sorts and dependency analysis, which can be treated in a more orthogonal manner, as we see in this chapter. The downsides of employing sorts for dependency analysis are described in detail in Mishra-Linger [2008].

Bernardy and Guilhem [2013] is very related to the calculi presented in this chapter. Bernardy and Guilhem [2013] use colors to erase terms while we use grades. Colors and grades both form a lattice structure and their usage in the respective type systems are quite similar. However, Bernardy and Guilhem [2013] use internalized parametricity to reason about erasure; so it is important that their type system is logically consistent. Our work, on the other hand, does not rely on the normalizing nature of the underlying type system; we take a direct route to analyzing erasure.

Lourenço and Caires [2015] design a calculus for tracking information flow in a dependent type system, similar to the calculi presented in this chapter. But Lourenço and Caires [2015] focus on more imperative features, like modeling of state, while we focus on irrelevance. A distinguishing feature of their system is that they allow security labels to depend upon terms, something that we don't attempt here.

## 3.7 Conclusion

We started with the aim of designing a dependent calculus that can analyze dependencies in general, and run-time and compile-time irrelevance in particular. Towards this end, we designed a simple dependency calculus, SDC, and then extended it to two dependent calculi, $\mathrm{DDC}^{\top}$ and DDC. $\mathrm{DDC}^{\top}$ can track run-time irrelevance while DDC can track both run-time and compile-time irrelevance, along with other dependencies. A novelty of DDC lies in its treatment of strong irrelevant $\Sigma$-types, a feature that often causes difficulty in the setting of compile-time irrelevance. Another novelty of DDC lies in its graded design, which enables a straightforward syntactic proof of noninterference. With these novel features, DDC can serve as a foundation for future dependency calculi in dependent type systems.

# Chapter 4

# Linearity Analysis in Pure Type Systems

Recall from Chapter 1 that the graded type systems [Brunel et al., 2014, Ghica and Smith, 2014, Petricek et al., 2014, etc.] based on BLL can analyze a wide variety of uses: linear use, no use, bounded use, unrestricted use, etc. However, as we pointed out, most of these systems are restricted to simple/polymorphic types. In this chapter, we extend them to dependent types.

In particular, this chapter makes the following contributions:

- Section 4.2 presents a graded simply-typed calculus, based on BLL, with standard algebraic types and a graded modal type. This system is not novel; instead, it establishes a foundation for the dependent system. However, even at this stage, we identify subtleties in the design space.
- Section 4.3 describes a heap-based operational semantics for the simply-typed calculus, inspired by Turner and Wadler [1999]. We prove the type system sound with respect to this instrumented semantics. An instrumented semantics is necessary to prove soundness because standard operational semantics does not track resources and as such, type safety does not imply that usage tracking is correct.
- Section 4.4 shows (a generalization of) the *single pointer property* for linear resources. The single pointer property says that a linear resource is referenced by precisely one pointer at run time. This property enables in-place update of linear resources.
- Sections 4.5 and 4.6 present the key contribution of this chapter: the language, GraD, that extends our ideas from simple to dependent types. We describe the design of the type system in Section 4.5 and extend the soundness proof with respect to heap semantics in Section 4.6.

## 4.1 The Algebra of Grades

The goal of a graded type system based on BLL is to track the demands that computations make on variables that appear in the context. In other words, the type system enables a static accounting of run-time resources 'used' in the evaluation of terms. Recall from Chapter 1 that this form of type system can track linear resources (resources used exactly once), bounded resources (resources used for a bounded number of times), unrestricted resources (resources used arbitrary number of times), etc. This general nature of the type system is made possible by its parametrization over an arbitrary preordered semiring of *grades* that model resource arithmetic. Below, we present formal definition and examples of preordered semirings.

Formally, a *semiring* is a set $Q$ with two binary operations, $\_+\_ : Q \times Q \to Q$ (addition) and $\_\cdot\_ : Q \times Q \to Q$ (multiplication), and two distinguished elements, 0 and 1, such that $(Q, +, 0)$ is a commutative monoid and $(Q, \cdot, 1)$ is a monoid; furthermore, multiplication is both left and right distributive over addition and 0 is an annihilator for multiplication. The elements of a semiring may be ordered. Such an order usually respects the binary operations of the semiring, meaning, for an order $<:$ on $Q$, if $q_1 <: q_2$, then for any $q \in Q$, we have, $q + q_1 <: q + q_2$ and $q \cdot q_1 <: q \cdot q_2$ and $q_1 \cdot q <: q_2 \cdot q$. A semiring with a preorder satisfying this condition is called a *preordered* semiring.

Next, we look at a few preordered semirings that are interesting from the perspective of usage tracking:

- The *trivial semiring* has a single element, and all operations just return that element. Our type system, when specialized to this semiring, degenerates to its standard ungraded counterpart.

- The *boolean semiring* has two elements, 0 and 1, with the property that $1 + 1 = 1$. A type system that draws grades from this semiring distinguishes between variables that are used (marked with 1) and ones that are unused (marked with 0). Note that in such a system, the grade 1 does *not* correspond to linear usage. This system does not count usage, but instead checks *whether* a variable is used or not.

  There are two different preorders that make sense for the boolean semiring. If we use the discrete order, then the type system would precisely track relevance: if a variable is marked with 0 in the context, we would know that the variable *must not* be used at run time, and if it is marked with 1, we would know that it *must* be used. On the other hand, if the preorder declares that $1 <: 0$, then we would still know that 0-marked variables are unused, but we would not know anything about the usage of 1-marked variables. Note here that, from a resource perspective, $q_1 <: q_2$ means that $q_1$ has more resource than $q_2$.

- The *linearity* and *affinity* preordered semirings, denoted $\mathcal{Q}_{\text{Lin}}$ and $\mathcal{Q}_{\text{Aff}}$ respectively, have three elements, 0, 1 and $\omega$, with addition and multiplication defined in the usual way after interpreting $\omega$ as 'greater than 1'. So, $1 + 1 = \omega + 1 = 1 + \omega = \omega + \omega = \omega$ and $\omega \cdot \omega = \omega$. The ordering in the linearity semiring is the reflexive closure of $\{(\omega, 0), (\omega, 1)\}$. The ordering

*(Grammar)*

| | | | |
|---|---|---|---|
| *types* | $A, B$ | ::= | $\mathbf{Unit} \mid {}^{q}A \to B \mid \Box^{q} A \mid A \otimes B \mid A \oplus B$ |
| *terms* | $a, b$ | ::= | $x \mid \lambda x{:}^{q} A.a \mid a\ b^{q}$ |
| | | $\mid$ | $\mathbf{unit} \mid \mathbf{let\,unit} = a\ \mathbf{in}\ b \mid \mathbf{box}_q\ a \mid \mathbf{let\,box}_q\ x = a\ \mathbf{in}\ b$ |
| | | $\mid$ | $(a, b) \mid \mathbf{let}\,(x, y) = a\,\mathsf{in}\,b$ |
| | | $\mid$ | $\mathbf{inj}_1\ a \mid \mathbf{inj}_2\ a \mid \mathbf{case}_q\ a\,\mathbf{of}\ b_1; b_2$ |
| *graded contexts* | $\Gamma$ | ::= | $\varnothing \mid \Gamma, x{:}^{q} A$ |
| *typing judgment* | | | $\boxed{\Gamma \vdash a : A}$ |

FIGURE 4.1: The simply typed graded $\lambda$-calculus

in the affinity semiring is the reflexive-transitive closure of $\{(\omega, 1), (1, 0)\}$. Over these semirings, a system can track linear and affine use respectively by marking linear/affine variables with 1 and unrestricted variables with $\omega$.

Preordered semirings have been used to track resource usage in many type systems [Abel and Bernardy, 2020, Atkey, 2018, Brunel et al., 2014, Gaboardi et al., 2016, Ghica and Smith, 2014, McBride, 2016, Orchard et al., 2019, Petricek et al., 2014, etc.] but there are some variations with respect to the formal requirements. For example: Brunel et al. [2014] require the underlying set (of the semiring) along with the order to form a bounded sup-semilattice while Abel and Bernardy [2020] define the order using an additional meet operation on the underlying set; McBride [2016] uses a hemiring (a semiring without 1) while Atkey [2018] uses a semiring with additional restrictions. Our theory is parametrized by a preordered semiring as defined in this section. We add additional constraints, as required, only while deriving specific properties in Section 4.4.

## 4.2 A BLL-Based Simple Type System

### 4.2.1 The Basics

Our goal is to design a *dependent* type system that analyzes resource usage. But, for simplicity, we start with a simple type system, similar to the one by Petricek et al. [2014]. The grammar for this system appears in Figure 4.1. It is parametrized over an arbitrary preordered semiring $(Q, 1, \cdot, 0, +, <:)$, with grades $q \in Q$.

The typing judgment for this system has the form: $\Gamma \vdash a : A$, where $\Gamma$ is a graded context. The judgment $x_1 :^{q_1} A_1, x_2 :^{q_2} A_2, \ldots, x_n :^{q_n} A_n \vdash b : B$ may be read as: one copy of $b$ uses $q_i$ copies of $x_i$, where $i = 1, 2, \ldots, n$.

A natural question to ask here is: what does using $q_i$ copies of $x_i$ mean? More importantly, what does variable 'use' mean? If we think of variables as pointers to data in memory, a 'use'

can be seen as a memory look-up of data. We look-up data when we reduce programs. But then, the number of memory look-ups depends on the reduction strategy employed. So when we say $b$ uses $q_i$ copies of $x_i$, we are surely assuming some reduction strategy. Indeed, this is the case.

The type system, presented below, implicitly assumes a call-by-name reduction strategy. Usage accounting would be quite different if we assume other reduction strategies, for example, a call-by-value or a call-by-need reduction strategy. To see why, let's consider some examples. The term $(\lambda x.42)\ z$ does not use $z$ in a call-by-name reduction but uses $z$ in a call-by-value reduction. Again, the term $\mathbf{let}\, x = 42\ \mathbf{in}\ \mathbf{let}\, y = x\ \mathbf{in}\ y + y$ uses two copies of $x$ in a call-by-name reduction but only one copy of $x$ in a call-by-need reduction. So then, $b$ uses $q_i$ copies of $x_i$ may be roughly understood as: during a call-by-name reduction, $b$ looks up $q_i$ times the data pointed to by $x_i$.

Next, we explore the algebraic structure of graded contexts.

A graded context, $\Gamma$, has two components: $\lfloor\Gamma\rfloor$, the underlying standard context and $\overline{\Gamma}$, the vector of grades in $\Gamma$. The operations and relation defined on grades, i.e. $+$, $\cdot$ and $<:$, can be lifted to graded contexts. We define $\Gamma_1 + \Gamma_2$ by point-wise addition of grades. However, such an addition makes sense only when $\lfloor\Gamma_1\rfloor = \lfloor\Gamma_2\rfloor$. Similarly, for any $q \in Q$, we define $q \cdot \Gamma$ by pre-multiplying every grade in $\Gamma$ by $q$. Further, if $\lfloor\Gamma_1\rfloor = \lfloor\Gamma_2\rfloor$, we say $\Gamma_1 <: \Gamma_2$ whenever every corresponding ordered pair of grades satisfy $<:$ relation. With these operations and relation defined on graded contexts, they form a nice algebraic structure. For any standard context $\Delta$, graded contexts $\Gamma$ that satisfy $\lfloor\Gamma\rfloor = \Delta$, form a preordered left $Q$-semimodule [Golan, 1999].

### 4.2.2 Type System

We now look at the type system. The typing rules appear inline below.

$$\textsc{ST-Var}\ \frac{}{0 \cdot \Gamma, x :^1 A \vdash x : A} \qquad \textsc{ST-Weak}\ \frac{\Gamma \vdash a : B}{\Gamma, x :^0 A \vdash a : B}$$

$$\textsc{ST-Unit}\ \frac{}{\varnothing \vdash \mathbf{unit} : \mathbf{Unit}} \qquad \textsc{ST-UnitE}\ \frac{\Gamma_1 \vdash a : \mathbf{Unit} \quad \Gamma_2 \vdash b : B \quad \lfloor\Gamma_1\rfloor = \lfloor\Gamma_2\rfloor}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\, \mathbf{unit} = a\ \mathbf{in}\ b : B}$$

In rule ST-Var, $x$ just needs one copy of $x$ and no copy of other variables; so $x$ appears at grade 1 and other variables at grade 0. Note also that $x :^1 A$ occurs last in the context. So we need an explicit weakening rule ST-Weak. Next, constants, like **unit**, do not need any resource, as we see in rule St-Unit. And to eliminate a term of type **Unit**, we just match it with **unit**. Since the elimination form requires the resources used by both the terms, we add the two contexts in the conclusion of rule ST-UnitE.

$$\textsc{ST-Lam}\ \frac{\Gamma, x{:}^q A \vdash b : B}{\Gamma \vdash \lambda x{:}^q A.b : ({}^q A \to B)} \qquad \textsc{ST-App}\ \frac{\Gamma_1 \vdash b : ({}^q A \to B) \quad \Gamma_2 \vdash a : A \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + q \cdot \Gamma_2 \vdash b\ a^q : B}$$

Now, in rule ST-Lam, $\lambda^q x.b$ needs $\Gamma$ and $q$ copies of its argument $x$; so $b$ needs $\Gamma$ and $q$ copies of $x$. In rule ST-App, $b$ needs $\Gamma_1$ and $q$ copies of its argument. Each copy of $a$, to-be argument of $b$, needs $\Gamma_2$. So $q$ copies of $a$ needs $q \cdot \Gamma_2$. Therefore, $b\ a^q$ needs $\Gamma_1 + q \cdot \Gamma_2$.

$$\textsc{ST-Box}\ \frac{\Gamma \vdash a : A}{q \cdot \Gamma \vdash \mathbf{box}_q\ a : \Box^q A} \qquad \textsc{ST-LetBox}\ \frac{\Gamma_1 \vdash a : \Box^q A \quad \Gamma_2, x{:}^q A \vdash b : B \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let\,box}_q\ x\ =\ a\ \mathbf{in}\ b : B}$$

Next, the graded modal type, $\Box^q A$, is introduced by the construct $\mathbf{box}_q\ a$, which uses the expression $q$ times to build the box. This box can then be passed around as an entity. When unboxed through rule ST-LetBox, the continuation has access to $q$ copies of the contents.

$$\textsc{ST-Pair}\ \frac{\Gamma_1 \vdash a : A \quad \Gamma_2 \vdash b : B \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash (a, b) : A \otimes B} \qquad \textsc{ST-LetPair}\ \frac{\Gamma_1 \vdash a : A_1 \otimes A_2 \quad \Gamma_2, x{:}^1 A_1, y{:}^1 A_2 \vdash b : B \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = a\,\mathsf{in}\,b : B}$$

Next, we have multiplicative products. The two components of a pair of this type do not share variable usages. Therefore, the introduction rule adds the two contexts together. Terms of this type must be eliminated via pattern-matching because both components should be used in the continuation. An elimination form that projects only one component of the pair would lose the usage constraints from the other component. Note that even though both components of the pair must be used exactly once, by nesting a modal type within the product, one can construct data structures with components of varying usage.

$$\textsc{ST-Inj1}\ \frac{\Gamma \vdash a : A_1}{\Gamma \vdash \mathbf{inj}_1\ a : A_1 \oplus A_2} \qquad \textsc{ST-Case}\ \frac{q <: 1 \quad \Gamma_1 \vdash a : A_1 \oplus A_2 \quad \Gamma_2 \vdash b_1 : {}^q A_1 \to B \quad \Gamma_2 \vdash b_2 : {}^q A_2 \to B \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{q \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_q\ a\ \mathbf{of}\ b_1; b_2 : B}$$

Finally, we have additive sums and case analysis. In rule ST-Case, the scrutinee and the branches may have different resource requirements. Note the side condition, $q <: 1$, in this rule. It reveals an interesting aspect of the reduction strategy implicit in our accounting. In a call-by-name reduction, $a$ will be first reduced to a value, irrespective of whether the branches require it or not. Now, suppose the branches don't require it, i.e. $q = 0$. But then, we have already 'used

$\boxed{a \rightsquigarrow a'}$ *(Small-step reduction)*

$$\text{S-AppBeta}\ \frac{}{(\lambda^q x.a)\ b^q \rightsquigarrow a\{b/x\}} \qquad \text{S-BoxBeta}\ \frac{}{\mathbf{let}\,\mathbf{box}_q\, x\, =\, \mathbf{box}_q\, a\ \mathbf{in}\ b \rightsquigarrow b\{a/x\}}$$

FIGURE 4.2: Small-step call-by-name reduction (excerpt)

up' resources in reducing $a$; we cannot go back and undo the usage. Therefore, we require that the branches 'use' $a$ at least once.

We also have a sub-usaging rule that allows us to provide more resources than is necessary. Note here that the relation $\Gamma_2 <: \Gamma_1$ means that $\Gamma_2$ has more resources than $\Gamma_1$.

$$\text{ST-Sub}\ \frac{\Gamma_1 \vdash a : A \qquad \Gamma_2 <: \Gamma_1}{\Gamma_2 \vdash a : A}$$

### 4.2.3 Type Soundness

Next, we look at some metatheoretic properties of the type system. The substitution lemma for the system is shown below. It is interesting because it needs to account for the grade of the variable being substituted. That grade is used to compute the resources required by the term after substitution.

**Lemma 4.1** (Substitution) If $\Gamma \vdash a : A$ and $\Gamma_1, x :^q A, \Gamma_2 \vdash b : B$, then $\Gamma_1 + q \cdot \Gamma, \Gamma_2 \vdash b\{a/x\} : B$.

Now, we consider the operational semantics of the language. Since the type system accounts usage with respect to a call-by-name semantics, we employ a call-by-name small-step reduction strategy for the language. All the step rules are standard other than the two rules that appear in Figure 4.2. These rules require that the grade in the introduction form matches with that in the corresponding elimination form.

With this operational semantics, a syntactic proof of type soundness follows in the usual manner, via the progress and preservation lemmas. This standard type-soundness theorem, however, is not very informative with regard to usage analysis because it does not really track usage. In order to address this issue, we turn to a heap-based semantics that accounts resource usage during computation. Our heap-based semantics is based on Launchbury [1993] and Turner and Wadler [1999].

## 4.3 Heap Semantics for Simple Type System

A heap semantics shows how a term evaluates when the free variables of the term are assigned other terms. These assignments are stored in a heap, represented here as an ordered list. We associate an *allowed usage*, basically a grade, to each assignment in the heap. We change these grades as reduction uses up resources. For example, a typical reduction goes like this:

$$
\begin{aligned}
& [x \overset{3}{\mapsto} 1, y \overset{1}{\mapsto} x + x](x + y) && \textit{look up value of } x\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{2}{\mapsto} 1, y \overset{1}{\mapsto} x + x]1 + y && \textit{look up value of } y\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{2}{\mapsto} 1, y \overset{0}{\mapsto} x + x]1 + (x + x) && \textit{look up value of } x\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{1}{\mapsto} 1, y \overset{0}{\mapsto} x + x]1 + (1 + x) && \textit{look up value of } x\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{0}{\mapsto} 1, y \overset{0}{\mapsto} x + x]1 + (1 + 1) && \textit{addition step} \\
\Rightarrow\ & [x \overset{0}{\mapsto} 1, y \overset{0}{\mapsto} x + x]3
\end{aligned}
$$

### 4.3.1 The Step Judgment

The reduction above is expressed informally as a sequence of heap and expression pairs. We formalize this relation using the following judgment.

$$[H]\, a \Rightarrow_S^r [H' ; \mathbf{u}' ; \Gamma']\, a'$$

The meaning of this judgment is that $r$ copies of the term $a$ use the resources of the heap $H$ and step to $r$ copies of the term $a'$, with $H'$ being the new heap. This judgment is sometimes written without $r$, which is assumed to be 1 in such cases. The judgment also maintains additional information, which we explain one by one.

Heap assignments are of the form $x \overset{q}{\mapsto} \Gamma \vdash a : A$, associating an *assignee variable*, $x$, with its *allowed usage*, $q$, and assignment, $a$. The *embedded context*, $\Gamma$, and type $A$ are there to assist in the proof of the soundness theorem. Like a graded context, a graded heap has two components: $\lfloor H \rfloor$, the underlying heap without the allowed usages and $\overline{H}$, the vector of allowed usages of the variables in $H$.

Because we use a call-by-name reduction, we don't evaluate the terms in the heap; we just modify the grades associated with the assignments as they are retrieved. Therefore, after any step, $H'$ will contain all the previous assignments of $H$, possibly at different allowed usages. However, a beta-reduction step may add new assignments to $H$. For example, $[\varnothing]\,(\lambda^1 x.x)\,\mathbf{unit} \Rightarrow [y \overset{1}{\mapsto} \mathbf{unit}]\, y$. Note the renaming of variable in this example. To allocate new variable names appropriately, we need to track the support set [Pitts, 2013], $S$, in the judgment; fresh names should be chosen avoiding the variables in this set. We keep track of the fresh names that are added to the heap along with the allowed usages of the corresponding assignments through the

*added context*, $\Gamma'$. Note that $\texttt{dom}\ H' = \texttt{dom}\ H \cup \texttt{dom}\ \Gamma'$, where $\texttt{dom}$ denotes the domain function. Next, we explain the role of $\mathbf{u}'$.

### 4.3.2 Non-determinism

Because we work with an arbitrary preordered semiring, possibly without subtraction, the reduction relation is non-deterministic. For example, consider a step $[x \overset{q}{\mapsto} a]x \Rightarrow [x \overset{q'}{\mapsto} a]a$, where $q = q' + 1$. Here, we are using $x$ once, so we need to reduce its usage by 1. But in some preordered semirings, there may exist multiple grades, $q'' \neq q'$, such that $q = q' + 1 = q'' + 1$. For example, in the linearity semiring, we have, $\omega = 1 + 1 = \omega + 1$. In such a case, $[x \overset{\omega}{\mapsto} a]x \Rightarrow [x \overset{1}{\mapsto} a]a$ and $[x \overset{\omega}{\mapsto} a]x \Rightarrow [x \overset{\omega}{\mapsto} a]a$.

The absence of subtraction also implies that given an initial heap and a final heap, we really don't know how much resources have been consumed by the computation. The only way to know this is to keep track of resources while they are being used. The amount of resources used up can be expressed as a grade vector $\mathbf{u}'$ called *consumption vector*, with its components showing usage, in any given step, of the corresponding variables in $H$. Note that even though $\mathbf{u}'$ records the usage of variables in $H$, its length is equal to that of $H'$: the usage corresponding to any newly-added definition is set to 0 in $\mathbf{u}'$.

### 4.3.3 Step Rules

With this understanding of the step judgment, let us now look at some of the step rules. First, the variable rule.

$$\textsc{Small-Var}\quad \frac{r <: 1}{[H_1, x \overset{(q+r)}{\mapsto} \Gamma \vdash a : A, H_2]\, x \Rightarrow^r_S [H_1, x \overset{q}{\mapsto} \Gamma \vdash a : A, H_2\,;\, \mathbf{0}^{|H_1|} \diamond r \diamond \mathbf{0}^{|H_2|}\,;\, \varnothing]\, a}$$

There are several points to note here. We reduce (look up) $r$ copies of $x$ and get $r$ copies of $a$. Starting with $(q + r)$ copies of $x$, we are left with $q$ copies of $x$. We note that $r$ copies of $x$ have been used in the consumption vector. Here, $|H|$ denotes the length of $H$ and $\mathbf{0}^n$ denotes a vector of 0's of length $n$ and $\mathbf{u}_1 \diamond \mathbf{u}_2$ denotes vector concatenation. Also, note the condition $r <: 1$. Because we are actually looking up $x$, we are using it at least once.

Next, we look at the rules for application.

$$\textsc{Small-AppL}\quad \frac{[H]\, a \Rightarrow^r_{S \cup \text{fv}\, b} [H'\,;\, \mathbf{u}'\,;\, \Gamma]\, a'}{[H]\, a\, b^q \Rightarrow^r_S [H'\,;\, \mathbf{u}'\,;\, \Gamma]\, a'\, b^q}$$

$$\textsc{Small-AppBeta}\quad \frac{x \textit{ fresh} \qquad a' = a\{x/y\}}{[H]\,(\lambda^q y.a)\; b^q \Rightarrow^r_S [H, x \overset{r\cdot q}{\mapsto} \Gamma \vdash b : A\,;\, \mathbf{0}^{|H|} \diamond 0\,;\, x{:}^{r\cdot q} A]\; a'}$$

The rule Small-AppL is straightforward; note, however, the change in support set. There are a few things to note in rule Small-AppBeta. First, it renames the bound variable $y$ with a fresh name $x$. Second, because we are reducing $r$ copies of the application, which is expected to use its argument $q$ times, the new assignment has allowed usage $r \cdot q$. Third, the type $A$ is arbitrary; the step relation is not concerned with appropriateness of typing.

Next, we look at the rule for subusage.

$$\textsc{Small-Sub}\quad \frac{[H_1]\; a \Rightarrow^r_S [H'\,;\, \mathbf{u}'\,;\, \Gamma]\; a' \qquad H_2 <: H_1}{[H_2]\; a \Rightarrow^r_S [H'\,;\, \mathbf{u}'\,;\, \Gamma]\; a'}$$

Rule Small-Sub reduces the allowed usages in the heap and then lets the term take a step. Note that the order relation on heaps is similar to that on graded contexts. More precisely, for heaps $H_1$ and $H_2$ such that $\lfloor H_1 \rfloor = \lfloor H_2 \rfloor$, we say $H_1 <: H_2$ whenever $\overline{H_1} <: \overline{H_2}$ (pointwise order).

Now that we have seen some step rules, let us look at the multi-step relation. The multi-step relation is the transitive closure of the single-step relation. The following rules build the relation:

$$\textsc{Multi-One}\quad \frac{[H]\; a \Rightarrow^r_S [H'\,;\, \mathbf{u}'\,;\, \Gamma]\; b}{[H]\; a \Rightarrow\!\!\!\Rightarrow^r_S [H'\,;\, \mathbf{u}'\,;\, \Gamma]\; b}$$

$$\textsc{Multi-Many}\quad \frac{[H]\; a \Rightarrow^r_S [H'\,;\, \mathbf{u}'\,;\, \Gamma_1]\; b_1 \qquad [H']\; b_1 \Rightarrow\!\!\!\Rightarrow^r_S [H''\,;\, \mathbf{u}''\,;\, \Gamma_2]\; b}{[H]\; a \Rightarrow\!\!\!\Rightarrow^r_S [H''\,;\, \mathbf{u}' \diamond \mathbf{0}^{|\Gamma_2|} + \mathbf{u}''\,;\, \Gamma_1, \Gamma_2]\; b}$$

The rule Multi-One is straightforward. There are two points to note in rule Multi-Many. First, we use the same grade $r$ in all the three judgments: $r$ copies of $a$ reduce to $r$ copies of $b_1$, which then reduce to $r$ copies of $b$. Second, the consumption vectors from the premise judgments are added up and the added contexts of new variables are concatenated.

At this point, let us step back for a moment and consider why we are designing this heap semantics. We designed this heap semantics so that we can model resource usage. Now that we have the semantics with us, how do we find out whether we have modeled resource usage correctly? A basic correctness principle for any accounting is the principle of conservation: that whatever we are accounting for does not come out of or vanish into thin air, that we see its progression. We define correct usage as usage that satisfies the principle of conservation. Next, we show that our semantics honors this principle.

### 4.3.4 Accounting of Resources

The step relation enforces fair usage of resources:

**Lemma 4.2** (Conservation) If $[H]\, a \twoheadrightarrow_S^r [H'; \mathbf{u}'; \Gamma']\, a'$, then $\overline{H} \diamond \overline{\Gamma'} <: \overline{H'} + \mathbf{u}'$.

Here, $\overline{H}$ represents the initial resources and $\overline{\Gamma'}$ represents the newly added resources; whereas $\overline{H'}$ represents the resources left and $\mathbf{u}'$ the resources that were consumed. So, the above lemma says that the initial resources concatenated with those that are added during evaluation, are equal to or more than the remaining resources plus those that were used up. Note that if we didn't allow sub-usaging i.e. if the preorder is discrete, the above lemma gives an exact equality. In such a scenario, the reduction relation enforces strict conservation of resources. More generally, this theorem states that we don't use more resources than what we are entitled to.

In this heap-based reduction, unlike substitution-based reduction, terms can 'get stuck' due to lack of resources. Let us look at an example:

$$
\begin{aligned}
& [x \overset{2}{\mapsto} 1, y \overset{1}{\mapsto} x + x](x + y) && \textit{look up value of } x\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{1}{\mapsto} 1, y \overset{1}{\mapsto} x + x]1 + y && \textit{look up value of } y\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{1}{\mapsto} 1, y \overset{0}{\mapsto} x + x]1 + (x + x) && \textit{look up value of } x\textit{, decrement its usage} \\
\Rightarrow\ & [x \overset{0}{\mapsto} 1, y \overset{0}{\mapsto} x + x]1 + (1 + x) && \textit{look up value of } x\textit{, stuck!}
\end{aligned}
$$

The evaluation gets stuck because the starting heap does not contain enough resources for the evaluation of the term. The term needs to use $x$ thrice; whereas the heap contains only two copies of $x$.

But this is not the only way in which an evaluation can run out of resources. Such a situation may also happen through 'unwise usage', even when the starting heap contains enough resources. For example, over the linearity semiring, the evaluation: $[x \overset{\omega}{\mapsto} 5]x + (x + x) \Rightarrow [x \overset{1}{\mapsto} 5]5 + (x + x) \Rightarrow [x \overset{0}{\mapsto} 5]5 + (5 + x)$ gets stuck because in the first step, $\omega$ was 'unwisely' split as $1 + 1$ instead of being split as $\omega + 1$.

Our aim, then, is to show that given a heap that contains enough resources, a well-typed term that is not a value, can always take a step such that the resulting heap contains enough resources for the evaluation of the resulting term. Next, we formalize what it means for a heap to contain enough resources to evaluate a term.

### 4.3.5 Heap Compatibility

The key idea behind the language design in this chapter is that, if the resources contained in a heap are judged to be 'right' for a term by the type system, then the evaluation of that term in

that heap does not get stuck. With the heap-based reduction rules enforcing fairness of usage, this would mean that the type system accounts resource usage properly.

The compatibility relation $H \Vdash \Gamma$, presented below, expresses the judgment that the heap $H$ contains enough resources to evaluate any term that type-checks in the graded context, $\Gamma$. A heap that is compatible with some context is called a *well-formed* heap.

$\boxed{H \Vdash \Gamma}$ *(Heap Compatibility)*

$$\textsc{Compat-Empty}\ \frac{}{\varnothing \Vdash \varnothing} \qquad \textsc{Compat-Cons}\ \frac{H \Vdash \Gamma_1 + (q \cdot \Gamma_2) \quad \Gamma_2 \vdash a : A}{H, x \overset{q}{\mapsto} \Gamma_2 \vdash a : A \Vdash \Gamma_1, x :^q A}$$

We now look at the compatibility rule Compat-Cons. This rule reminds us of the substitution lemma 4.1. It loads $q$ potential single-substitutions into the heap and lets the context use the variable $q$ times. So, in a way, it is converse of the substitution lemma. Formally, we have:

**Lemma 4.3** (Multi-substitution) If $H \Vdash \Gamma$ and $\Gamma \vdash a : A$, then $\varnothing \vdash a\{H\} : A$.

Here $a\{H\}$ denotes the term obtained by substituting in $a$, in reverse order, the definitions in $H$.

Next, we look at an example derivation of a compatible heap-context pair. (We simplify the judgments by omitting some arguments.)

**Example 4.1.**

$$\dfrac{\dfrac{\dfrac{\varnothing \vdash \varnothing \qquad \varnothing \vdash 1 : \mathbf{Int}}{x_1 \overset{7}{\mapsto} 1 \vdash x_1 :^7 \mathbf{Int}} \qquad x_1 :^2 \mathbf{Int} \vdash x_1 + x_1 : \mathbf{Int}}{x_1 \overset{7}{\mapsto} 1, x_2 \overset{3}{\mapsto} x_1 + x_1 \vdash x_1 :^1 \mathbf{Int}, x_2 :^3 \mathbf{Int}} \qquad x_1 :^1 \mathbf{Int}, x_2 :^2 \mathbf{Int} \vdash x_1 + (x_2 + x_2) : \mathbf{Int}}{x_1 \overset{7}{\mapsto} 1, x_2 \overset{3}{\mapsto} x_1 + x_1, x_3 \overset{1}{\mapsto} x_1 + (x_2 + x_2) \vdash x_1 :^0 \mathbf{Int}, x_2 :^1 \mathbf{Int}, x_3 :^1 \mathbf{Int}}$$

The context $x_1 :^7 \mathbf{Int}$ splits its resources among definitions $x_2 \mapsto x_1 + x_1$ (thrice) and $x_3 \mapsto x_1 + (x_2 + x_2)$ (once). We see that the heap keeps a record, in the form of allowed usages, of how the context gets split.

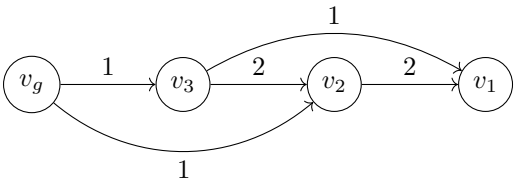
The compatibility relation is crucial to our development. So we explore it in more detail below.

### 4.3.6 Graphical and Algebraic Views of the Heap

A heap can be viewed as a memory graph where the assignee variables correspond to memory locations and the assigned terms to data stored in those locations. The allowed usage then, is the number of references to that location. This gives us a graphical view of the heap.

A well-formed heap $H$, where $H \Vdash \Gamma$, can be viewed as a weighted directed acyclic graph $G_{H,\Gamma}$. Below, we describe this graph. Let $H$ contain $n$ assignments with the $j^{\text{th}}$ one being $x_j \overset{q_j}{\mapsto} \Gamma_j \vdash a_j : A_j$. Then, $G_{H,\Gamma}$ is a directed acyclic graph with $(n+1)$ nodes, $n$ nodes corresponding to the $n$ variables in $H$ and one extra node for $\Gamma$, referred to as the ground node. Let $v_j$ be the node corresponding to $x_j$ and $v_g$ be the ground node. For $x_i :^{q_{ji}} A_i \in \Gamma_j$ (where $i < j$), add an edge with weight $w(v_j, v_i) := q_{ji}$ from $v_j$ to $v_i$. (Note that $\Gamma_j$ only contains variables $x_1$ through $x_{j-1}$.) We do this for all nodes, including $v_g$. This gives us a directed acyclic graph with the topological ordering $v_g, v_n, v_{n-1}, \ldots, v_2, v_1$.

For example 4.1, we have the following memory graph (any edge with weight 0 is omitted):




For a well-formed heap, we can express the allowed usages of the assignee variables in terms of the edge weights of the memory graph: Let us define the length of a path to be the product of the weights along the path. Then, the allowed usage of a variable is the sum of the lengths of all paths from the source node to the node corresponding to that variable. Note that this is so for the example graph.

A path $p$ from $v_g$ to $v_j$ represents a chain of references, with the last one being pointed at $v_j$. The length of $p$ shows how many times this path is used to reference $v_j$. The sum of the lengths of all the paths from $v_g$ to $v_j$ then gives a (static) count of the total number of times location $v_j$ is referenced. And this is equal to $q_j$, the allowed usage of the assignment for $v_j$ in the heap. This means that the allowed usage of an assignment is equal to the (static) count of the number of times the concerned location is referenced. We call this property count balance. Below, we present an algebraic formalization of this property of well-formed heaps.

For a well-formed heap $H$ containing $n$ assignments of the form $x_i \overset{q_i}{\mapsto} \Gamma_i \vdash a_i : A_i$, we write $\langle H \rangle$ to denote the $n \times n$ matrix whose $i^{\text{th}}$ row is $\overline{\Gamma_i} \diamond \mathbf{0}$. We call $\langle H \rangle$ the transformation matrix corresponding to $H$. We use $\mathbf{0}$ to denote a row vector of 0s (of length $n$, where $n$ is clear from the context) and use $\mathbf{0}^\intercal$ to denote a column vector of 0s. The transformation matrix for example 4.1 is:

$$\begin{pmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 1 & 2 & 0 \end{pmatrix}$$

For a well-formed heap $H$, the matrix $\langle H \rangle$ is strictly lower triangular. Observe that this is also the adjacency matrix of the memory graph, excluding node $v_g$. The strict lower triangular

property of the matrix corresponds to the acyclicity of the graph. With the matrix operations over a semiring defined in the usual way, the count balance property can be stated as:

**Lemma 4.4** (Count Balance) If $H \Vdash \Gamma$, then $\overline{H} = \overline{H} \times \langle H \rangle + \overline{\Gamma}$.

For example 4.1, we can check that $\overline{H} = \begin{pmatrix} 7 & 3 & 1 \end{pmatrix}$ satisfies the above equation. We explain the intuition behind this equation. For a node $v_i$ in $G_{H,\Gamma}$, we have, $q_i = \Sigma_j \, q_j w(v_j, v_i) + w(v_g, v_i)$. The right-hand side of this equation gives a static estimate of demand, the amount of resources we shall need while the left-hand side gives a static estimate of supply, the amount of resources we shall have. So $H \Vdash \Gamma$ is a static guarantee that the heap $H$ shall supply the resource demands of the context $\Gamma$.

Therefore, if $H \Vdash \Gamma$ and $\Gamma \vdash a : A$, we should be able to evaluate $a$ in $H$ without running out of resources. This is the gist of the soundness theorem.

### 4.3.7 Soundness

**Theorem 4.5** (Soundness) If $H \Vdash \Gamma$ and $\Gamma \vdash a : A$ and $S \supseteq \mathsf{dom}\,\Gamma$, then either $a$ is a value or there exists $\Gamma'$, $H'$, $\mathbf{u}'$, $\Gamma_0$ such that:

- $[H]\, a \Rightarrow_S [H' ; \mathbf{u}' ; \Gamma_0]\, a'$
- $H' \Vdash \Gamma'$
- $\Gamma' \vdash a' : A$
- $\overline{\Gamma} \diamond \mathbf{0} + \mathbf{u}' \times \langle H' \rangle + \mathbf{0} \diamond \overline{\Gamma_0} \;<:\; \overline{\Gamma'} + \mathbf{u}' + \mathbf{0} \diamond \overline{\Gamma_0} \times \langle H' \rangle$

The soundness theorem states that our computations can go forward with the available resources without ever getting stuck. Note that as the term $a$ steps to $a'$, the typing context changes from $\Gamma$ to $\Gamma'$. This is to be expected because during the step, resources from the heap may have been consumed or new resources may have been added. For example, $[x \overset{1}{\mapsto} \mathbf{unit}]x \Rightarrow [x \overset{0}{\mapsto} \mathbf{unit}]\mathbf{unit}$ and $x :^1 \mathbf{Unit} \vdash x : \mathbf{Unit}$ while $x :^0 \mathbf{Unit} \vdash \mathbf{unit} : \mathbf{Unit}$. Though the typing context may change, the new context, which type-checks the reduct, must be compatible with the new heap. This means that we can apply the soundness theorem again and again until we reach a value. At every step of the evaluation, the dynamics of the language aligns perfectly with its statics. Graphically, as the evaluation progresses, the weights in the memory graph change but the count balance property is maintained.

Furthermore, the old and new contexts are related according to the fourth clause of the theorem. For the moment being, let the preorder be discrete. Then, we have:

$$\begin{array}{rcccc} & \overline{\Gamma'} & + & \mathbf{u}' & + & \mathbf{0} \diamond \overline{\Gamma_0} \times \langle H' \rangle \\ = & \overline{\Gamma} \diamond \mathbf{0} & + & \mathbf{u}' \times \langle H' \rangle & + & \mathbf{0} \diamond \overline{\Gamma_0} \end{array}$$

We can understand this equation through the following analogy. The contexts can be seen engaged in a transaction with the heap. The heap pays the context $\mathbf{0} \diamond \overline{\Gamma_0}$ and gets $\mathbf{0} \diamond \overline{\Gamma_0} \times \langle H' \rangle$ resources in return. The context pays the heap $\mathbf{u}'$ and gets $\mathbf{u}' \times \langle H' \rangle$ resources in return. The equation is the "balance sheet" of this transaction.

For an arbitrary preorder, the transaction gets skewed in favor of the heap; meaning, the context gets less from the heap for what it pays. This is so because the heap contains more resources than is necessary; so it may 'throw away' extra resources.

This soundness theorem subsumes standard type soundness: we can derive standard preservation and progress lemmas from the above theorem. We can also apply the above theorem to derive some useful properties about usage.

## 4.4 Applications

### 4.4.1 No Use

Till now, we have developed the theory over an arbitrary preordered semiring. But an arbitrary semiring is too general a structure to reason about usage. For instance, the set $\{0, 1\}$ with $1 + 1 = 0$ and all other operations defined in the usual way is also a semiring. But such a semiring does not capture our notion of usage since grades 0 and 1 are supposed to represent no usage and some usage respectively.

For grade 0 to represent no usage in a preordered semiring $Q$, the equation $0 <: q + 1$ must have no solution in $Q$. We shall call preordered semirings that satisfy this condition zero-unusable. Next, we relate grade 0 and no usage in zero-unusable semirings.

**Lemma 4.6** In a zero-unusable semiring, if $[H]\, a \Rightarrow_S [H'\,;\, \mathbf{u}'\,;\, \Gamma_4]\, a'$ and $x_i \overset{0}{\mapsto} \Gamma_i \vdash a_i : A_i \in H$, then the component $\mathbf{u}'(x_i) = 0$ and $x_i \overset{0}{\mapsto} \Gamma_i \vdash a_i : A_i \in H'$.

We see above that variables with grade 0 cannot be referenced during computation. Further, such variables always remain at grade 0. Now, if they cannot be referenced, what they contain should not matter. In other words, 0-graded variables should not affect the result of computation. Therefore, two initial heap configurations that differ only in the assignments of some 0-graded variables should produce identical results. The following lemma shows that this is indeed the case.

**Lemma 4.7** Let $H_{i\,1} = x_i \overset{0}{\mapsto} \Gamma_1 \vdash a_1 : A_1$ and $H_{i\,2} = x_i \overset{0}{\mapsto} \Gamma_2 \vdash a_2 : A_2$. Then, in a zero-unusable semiring, if $[H_1, H_{i\,1}, H_2]\, b \Rightarrow_{S \cup \mathrm{fv}\, a_2} [H_1', H_{i\,1}, H_2'\,;\, \mathbf{u}'\,;\, \Gamma_0]\, b'$,
then $[H_1, H_{i\,2}, H_2]\, b \Rightarrow_{S \cup \mathrm{fv}\, a_1} [H_1', H_{i\,2}, H_2'\,;\, \mathbf{u}'\,;\, \Gamma_0]\, b'$.

### 4.4.2 Linear Use

Next, we relate grade 1 and linear use. Similar to the case of grade 0 and no use, grade 1 does not always represent linear use. We need to put the following restrictions on the parametrizing preordered semiring $(Q, +\cdot, 0, 1, <:)$ to ensure this.

- For $q_1, q_2 \in Q$, the equation $q_1 + q_2 = 0$ implies $q_1 = q_2 = 0$. This property is called zerosumfree [Golan, 1999].
- For $q_1, q_2 \in Q$, the equation $q_1 \cdot q_2 = 0$ implies $q_1 = 0$ or $q_2 = 0$. This property is called entire [Golan, 1999].
- For $q_1, q_2 \in Q$, the equation $q_1 + q_2 = 1$ implies $q_1 = 1$ and $q_2 = 0$ or vice-versa.
- For $q_1, q_2 \in Q$, the equation $q_1 \cdot q_2 = 1$ implies $q_1 = q_2 = 1$.
- 0 and 1, where $0 \neq 1$, are maximal elements with respect to $<:$ relation.

We shall call preordered semirings that satisfy the above constraints one-linear. The linearity semiring is a one-linear semiring. Observe that one-linear semirings are also zero-unusable. For one-linear semirings, we can state the single pointer property [Turner and Wadler, 1999] as:

**Lemma 4.8** In a one-linear semiring: If $H \Vdash \Gamma$ and $x_i \overset{1}{\mapsto} \Gamma_i \vdash a_i : A_i \in H$, then in $G_{H,\Gamma}$, there is only one path from the ground node to $v_i$ and all the weights on that path are 1.

In words, the above theorem says that there is one and only one way to reference a linear resource; any resource along that way also has a single pointer to it. This property enables safe in-place update of linear resources.

This completes our discussion on the BLL-based simple type system. Now, we move on to the BLL-based dependent calculus, GRAD.

## 4.5 BLL-Based Dependent Type System

### 4.5.1 The Basics

What distinguishes the dependent system from its simple counterpart is that term variables can occur in types. When variables occur both in terms and types, counting their usage is not straightforward. Recall that in Chapter 1, we discussed three valid approaches to accounting resource usage in a dependent type system. The approach we follow in this chapter treats types and terms uniformly and counts usage only in the expression that appears as a term in a given typing judgment. The variable rule T-VAR of the dependent system makes this point clear:

$$\textsc{T-Var}\ \frac{\Gamma \vdash A : s}{0 \cdot \Gamma, x :^{1} A \vdash x : A}$$

We account for the resource usage of $A$ in $\Gamma$. But when $A$ appears as the type of $x$, we zero out the usage of $A$ and take into account only the usage of $x$, which is one copy of $x$. We use this accounting principle in GRAD.

GRAD is a Pure Type System (PTS) characterized by a tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, where $s \in \mathcal{S}$ are sorts, $\mathcal{A}$ is a set of axioms and $\mathcal{R}$ is a ternary relation between sorts. Like the simple system, GRAD has functions, multiplicative products, additive sums and unit. However, the types are now dependent — we have Π-type, Σ-type and disjoint sum type. Note that we do not have an explicit modal type; we can encode modal type using Σ-type and **Unit**. Formally, the types and terms of GRAD are as follows:

$$\begin{array}{lrcl} \textit{terms}, \textit{types} & a, b, A, B & ::= & s \mid x \\ & & \mid & \mathbf{Unit} \mid \mathbf{unit} \mid \mathbf{let\,unit}\ =\ a\ \mathbf{in}\ b \\ & & \mid & \Pi x :^{q} A.B \mid \lambda x :^{q} A.a \mid a\ b^{q} \\ & & \mid & \Sigma x :^{q} A.B \mid (a^{q}, b) \mid \mathbf{let}\,(x^{q}, y)\ =\ a\ \mathbf{in}\ b \\ & & \mid & A \oplus B \mid \mathbf{inj}_1\ a \mid \mathbf{inj}_2\ a \mid \mathbf{case}_q\ a\ \mathbf{of}\ b_1; b_2 \end{array}$$

### 4.5.2 Type System

Let us look at some typing rules. The rules T-PI and T-LAM are interesting.

$$\textsc{T-Pi}\ \frac{\Gamma_1 \vdash A : s_1 \qquad \Gamma_2, x :^{r} A \vdash B : s_2 \qquad \mathcal{R}(s_1, s_2, s_3) \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash \Pi x :^{q} A.B : s_3}$$

$$\textsc{T-Lam}\ \frac{\Gamma_1, x :^{q} A \vdash b : B \qquad \Gamma_2 \vdash \Pi x :^{q} A.B : s \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 \vdash \lambda x :^{q} A.b : \Pi x :^{q} A.B}$$

The first thing to note in rule T-PI is that the grade annotation $q$ on the bound variable $x$ in the type may be different from the usage $r$ of the variable $x$ in the body $B$ of the type. The grade $q$ is meant for tracking the usage of the argument in the body of a function having this type, as we see in rule T-LAM, and this usage may have no relation to the usage $r$ of $x$ in the body of the type itself. This difference between $q$ and $r$ allows GRAD to represent parametric polymorphism by marking type arguments with usage 0. For example, the System F type, $\forall \alpha.\alpha \to \alpha$, can be expressed in GRAD as $\Pi x :^{0} \texttt{Type}.^{1} x \to x$. This type is well-formed because, even though the annotation on the variable $x$ is 0, rule T-PI allows $x$ to be used any number of times in the body of the type.

Some versions of irrelevant quantifiers in type theories constrain $r$ to be equal to $q$ [Abel and Scherer, 2012]. By coupling the usage of variables in the body of the lambda with that of the Π,

these systems rule out the representation of polymorphic types, such as the one shown above. In our language, we can model this more restricted form of quantifier with the assistance of the box modality. If, instead of using the type $\Pi x :^0 A.B$, we use the type $\Pi x :^1 \Box^0 A.B$, the unboxed argument cannot appear in a relevant position within $B$ because terms of type $\Box^0 A$, upon unboxing, must be used 0 times only.

It is this distinction between the types $\Pi x :^0 A.B$ and $\Pi x :^1 (\Box^0 A).B$ (and a similar distinction between $\Sigma x{:}^0 A.B$ and $\Sigma x{:}^1(\Box^0 A).B$) that motivates our inclusion of the usage annotation on the $\Pi$ and $\Sigma$ types directly. In the simple type system, we can derive usage-annotated functions from linear functions and the box modality: there is no need to annotate the arrow with any quantity other than 1. But here, due to dependency, we cannot have parametrically polymorphic types without explicit annotation on $\Pi$-types.

Now, we look at $\Sigma$-types.

T-Sigma

$$\frac{\Gamma_1 \vdash A : s_1 \qquad \Gamma_2, x :^r A \vdash B : s_2 \qquad \mathcal{R}(s_1, s_2, s_3) \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash \Sigma x{:}^q A.B : s_3}$$

T-Pair

$$\frac{\Gamma_1 \vdash a : A \qquad \Gamma_2 \vdash b : B\{a/x\} \qquad \Gamma_3, x :^r A \vdash B : s \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor = \lfloor \Gamma_3 \rfloor}{q \cdot \Gamma_1 + \Gamma_2 \vdash (a^q, b) : \Sigma x{:}^q A.B}$$

T-LetPair

$$\frac{\Gamma_1 \vdash a : \Sigma x{:}^q A_1.A_2 \qquad \Gamma_2, x :^q A_1, y :^1 A_2 \vdash b : B\{(x,y)/z\} \qquad \Gamma_3, z :^r (\Sigma x{:}^q A_1.A_2) \vdash B : s \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor = \lfloor \Gamma_3 \rfloor}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x^q, y) \;=\; a \;\mathbf{in}\; b : B\{a/z\}}$$

The rule T-Sigma is similar to rule T-Pi. The grade $q$ on the bound variable $x$ tracks the number of copies of the first component in a pair having this type, as we see in rule T-Pair. This grade has no relation to the usage $r$ of $x$ in the body of the type itself. The rule T-LetPair is similar to its counterpart in the simple system; however, it implements a dependent pattern-matching.

Recall that $\Sigma$-types that are eliminated via pattern-matching are called weak $\Sigma$-types as opposed to strong $\Sigma$-types that are eliminated via projections. By this definition, $\Sigma$-types in Grad are weak $\Sigma$-types. There does not seem to be a straightforward way to include strong $\Sigma$-types in Grad. The reason behind this is that the pairing rule *adds up* the resources used by both the components making it difficult for any projection rule to figure out what part of that sum is used by any single projection. However, if the pairing rule does not add up the resources needed by the individual components and instead, lets both the components share the same set of resources, then those pairs can be eliminated by projections. But the downside of such pairs is that both the components *must use* the same set of resources.

The conversion rule of GraD, shown below, uses the definitional equality relation. Definitional equality for GraD is essentially $\beta$-equivalence, for call-by-name $\beta$-reduction. We show some of the equality rules below. They are mostly standard. However, the congruence rules, for example, rules Eq-PiCong and Eq-AppCong, need to check that the grade annotations on the terms being equated match up.

$$\textsc{T-Conv}\quad \frac{\Gamma_1 \vdash a : A \qquad \Gamma_2 \vdash B : s \qquad A =_\beta B \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 \vdash a : B}$$

$$\textsc{Eq-PiCong}\quad \frac{A_1 =_\beta A_2 \qquad B_1 =_\beta B_2}{\Pi x :^r A_1 . B_1 =_\beta \Pi x :^r A_2 . B_2} \qquad \textsc{Eq-AppCong}\quad \frac{b_1 =_\beta b_2 \qquad a_1 =_\beta a_2}{b_1\ {a_1}^r =_\beta b_2\ {a_2}^r}$$

Now that we have seen the type system, let us look at the metatheory.

### 4.5.3 Metatheory

We present the substitution and weakening lemmas for the calculus below. With respect to resource analysis, these lemmas are similar to their counterparts in the simply-typed system.

**Lemma 4.9** (Substitution) If $\Gamma \vdash a : A$ and $\Gamma_1, x :^q A, \Gamma_2 \vdash b : B$ then $(\Gamma_1 + q \cdot \Gamma), \Gamma_2\{a/x\} \vdash b\{a/x\} : B\{a/x\}$.

**Lemma 4.10** (Weakening) If $\Gamma_1, \Gamma_2 \vdash a : A$ and $\Gamma_3 \vdash B : s$ then $\Gamma_1, x :^0 B, \Gamma_2 \vdash a : A$.

Next, with a small-step call-by-name reduction relation that is same as that of the simply-typed system, we have the following type soundness theorem for GraD.

**Theorem 4.11** (Preservation) If $\Gamma \vdash a : A$ and $a \rightsquigarrow a'$ then $\Gamma \vdash a' : A$.

**Theorem 4.12** (Progress) If $\varnothing \vdash a : A$ then either $a$ is a value or there exists some $a'$ such that $a \rightsquigarrow a'$.

Now, similar to the simply-typed system, we develop a heap semantics for GraD.

## 4.6 Heap Semantics for GraD

The presence of dependent types causes one issue with the heap semantics: because substitutions are delayed through the heap, the terms and their types can 'get out of sync'.

For example, if we apply the polymorphic identity function, $\lambda x :^0 s . \lambda y :^1 x . y$, to some type argument, say **Unit**, then the result $(\lambda x :^0 s . \lambda y :^1 x . y)\,\mathbf{Unit}$ should have type $\Pi y :^1 \mathbf{Unit} . \mathbf{Unit}$.

By the rule Small-AppBeta, $[\varnothing]\,(\lambda x :^0 s.\lambda y :^1 x.y)\,\mathbf{Unit} \Rightarrow [x \overset{0}{\mapsto} \mathbf{Unit}]\,\lambda y :^1 x.y$. The term $\lambda y :^1 x.y$ has type $\Pi y :^1 x.x$. But since $x = \mathbf{Unit}$, we see that $\Pi y :^1 x.x = \Pi y :^1 \mathbf{Unit}.\mathbf{Unit}$. As such, $\lambda y :^1 x.y$ can also be assigned the type $\Pi y :^1 \mathbf{Unit}.\mathbf{Unit}$. So to align the types of the redex and the reduct, we need to know about the new assignments loaded into the heap. This issue did not exist in the simple setting because types did not depend on term variables. Note that this is not a usage-related issue, any heap-based reduction that delays substitution will need to address it while proving soundness. The good news is that it can be resolved with a simple extension to the type system.

### 4.6.1 A Dependently-Typed Language with Definitions

We extend our contexts with definitions that mimic delayed substitutions. These definitions are used *only* in deriving type equalities. From the type system perspective, they are essentially a bookkeeping device added to enable reasoning with respect to the heap semantics.

$$\textit{graded contexts}, \quad \Gamma \quad ::= \quad \varnothing \mid \Gamma, x :^q A \mid \Gamma, x = a :^q A$$

Along with this extension to the context, we modify the conversion rule and add two new typing rules to the system, as shown below. (In rule T-Conv-Def, $A\{\Gamma\}$ denotes the type obtained by substituting in $A$, in reverse order, the definitions of the variables in $\Gamma$.)

$\boxed{\Gamma \vdash a : A}$ *(Typing rules for dependent system with definitions)*

T-Conv-Def
$$\frac{\Gamma_1 \vdash a : A \qquad \Gamma_2 \vdash B : s \qquad A\{\Gamma_1\} =_\beta B\{\Gamma_1\} \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 \vdash a : B}$$

T-Var-Def
$$\frac{\Gamma \vdash a : A}{0 \cdot \Gamma, x = a :^1 A \vdash x : A}$$

T-Weak-Def
$$\frac{\Gamma_1 \vdash a : A \qquad \Gamma_2 \vdash b : B \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1, y = b :^0 B \vdash a : A}$$

The definitions act like usual variable assumptions: rules T-Var-Def and T-Weak-Def mirror rules T-Var and T-Weak respectively. Note that definitions are applied only in the conversion rule T-Conv-Def that substitutes these definitions before comparing for $\beta$-equivalence. The modified conversion rule allows the term $\lambda y :^1 x.y$ to be be assigned the type $\Pi y :^1 \mathbf{Unit}.\mathbf{Unit}$ in a context that defines $x$ to be **Unit**.

The extended type system enjoys the same syntactic soundness properties mentioned in Section 4.5.3. Furthermore, because definitions act only on types, they do not add extra resource demands to any typing derivation. As a result, we can always update a normal assumption with an appropriate definition in any typing judgment, as we see in the lemma below. In this regard, the resources used by the definiens ($\Gamma$ below) do not matter.

**Lemma 4.13** If $\Gamma_1, x :^q A, \Gamma_2 \vdash b : B$ and $\Gamma \vdash a : A$, then $\Gamma_1, x = a :^q A, \Gamma_2 \vdash b : B$.

We can also weaken contexts with new (unused) definitions in any typing judgment, analogous to lemma 4.10.

**Lemma 4.14** (Weakening with Definitions) If $\Gamma_1, \Gamma_2 \vdash b : B$ and $\Gamma \vdash a : A$ then $\Gamma_1, x = a :^0 A, \Gamma_2 \vdash b : B$.

Now, because we have modified contexts to include definitions, we need to modify the heap reduction and compatibility relations. However, the modifications required are minor. As far as the reduction relation is concerned, only the rules that load new assignments into the heap need a change: the added context of new variables should now remember their assignments. For example, the rule Small-AppBeta should be:

Small-DAppBeta

$$\frac{x \textit{ fresh} \qquad a' = a\{x/y\}}{[H]\,(\lambda^q y.a)\; b^q \Rightarrow^r_S [H, x \overset{r\cdot q}{\mapsto} \Gamma \vdash b : A\,;\, \mathbf{0}^{|H|} \diamond 0\,;\, x = b :^{r\cdot q} A]\; a'}$$

As far as the compatibility relation is concerned, rule Compat-Cons also needs to ensure that the contexts remember the definitions of their domain variables. With this modification, we have the following interesting property: if $H \Vdash \Gamma$ then $b\{H\} = b\{\Gamma\}$ for any term $b$.

$\boxed{H \Vdash \Gamma}$ *(Compatibility with definitions)*

Compat-ConsDef

$$\frac{H \Vdash \Gamma_1 + (q \cdot \Gamma_2) \qquad \Gamma_2 \vdash a : A}{H, x \overset{q}{\mapsto} \Gamma_2 \vdash a : A \Vdash \Gamma_1, x = a :^q A}$$

These are all the changes we need in reduction and compatibility relations. Now, we describe the changes needed in lemmas/theorems featuring these relations. Since we just modify added contexts in the heap reduction relation and given that added contexts do not play a major role in the relation, the previously stated lemmas (for example, lemma 4.2) regarding this relation hold. But with regard to the compatibility relation, we need to modify the statement of a lemma: the multi-substitution property (lemma 4.3) now needs to substitute into the type, in addition to the term.

**Lemma 4.15** (Multi-substitution) If $H \Vdash \Gamma$ and $\Gamma \vdash a : A$, then $\varnothing \vdash a\{H\} : A\{H\}$.

Now that we have modified all the relations and theorems to accommodate definitions, let us reflect how this extended system with definitions relate to the original one without definitions. Towards this end, we relate the original typing and heap compatibility judgments to their extended counterparts. For the sake of distinction, we denote the original relations by $\vdash_o$ and $\Vdash_o$

respectively. Now, for $H \Vdash_o \Gamma$, let $\Gamma_H$ denote $\Gamma$ with the variables defined according to (assignments in) $H$. Also, let $H_H$ denote $H$ with the variables in the embedded contexts in $H$ defined according to $H$. Then, we have:

**Lemma 4.16** (Elaboration) If $\Gamma \vdash_o a : A$ and $H \Vdash_o \Gamma$, then $\Gamma_H \vdash a : A$ and $H_H \Vdash \Gamma_H$.

The above theorem says that typing and compatibility judgments in the original system can be elaborated to typing and compatibility judgments in the extended system. Thus, if the extended system is sound with respect to usage tracking, then so is the original system. Next, we show that the extended system is indeed sound.

### 4.6.2 Proof of the Heap Soundness Theorem

The soundness theorem for the calculus follows as an instance of the invariance lemma below. This lemma provides a strong enough hypothesis for the induction to go through.

**Lemma 4.17** (Invariance) If $H \Vdash \Gamma_0 + q \cdot \Gamma$ and $\Gamma \vdash a : A$ and $q <: 1$ and $S \supseteq \mathsf{dom}\,\Gamma$, then either $a$ is a value or there exists $\Gamma'$, $H'$, $\mathbf{u}'$, $\Gamma_1$ and $a'$ such that:

- $[H]\, a \Rightarrow_S^q [H'; \mathbf{u}'; \Gamma_1]\, a'$
- $H' \Vdash (\Gamma_0, 0 \cdot \Gamma_1) + q \cdot \Gamma'$
- $\Gamma' \vdash a' : A$
- $q \cdot (\overline{\Gamma} \diamond \mathbf{0}) + \mathbf{u}' \times \langle H' \rangle + \mathbf{0} \diamond \overline{\Gamma_1} <: q \cdot \overline{\Gamma'} + \mathbf{u}' + \mathbf{0} \diamond \overline{\Gamma_1} \times \langle H' \rangle$
- $\mathsf{dom}\,\Gamma_1$ is disjoint from $S$

**Theorem 4.18** (Soundness) If $H \Vdash \Gamma$ and $\Gamma \vdash a : A$ and $S \supseteq \mathsf{dom}\,\Gamma$, then either $a$ is a value or there exists $\Gamma'$, $H'$, $\mathbf{u}'$, $\Gamma_1, A'$ such that:

- $[H]\, a : A \Rightarrow_S [H'; \mathbf{u}'; \Gamma_1]\, a' : A'$
- $H' \Vdash \Gamma'$
- $\Gamma' \vdash a' : A'$
- $\overline{\Gamma} \diamond \mathbf{0} + \mathbf{u}' \times \langle H' \rangle + \mathbf{0} \diamond \overline{\Gamma_1} <: \overline{\Gamma'} + \mathbf{u}' + \mathbf{0} \diamond \overline{\Gamma_1} \times \langle H' \rangle$

The soundness theorem shows that usage analysis in GRAD is correct. We can use this theorem to prove the usual preservation and progress lemmas. We can also use this theorem to reason about no use and linear use, as discussed in Section 4.4.

This soundness theorem is similar in spirit to theorems showing correctness of usage in BLL-based type systems via operational methods [Abel and Bernardy, 2020, Brunel et al., 2014, etc]. But this theorem can be proved by simple induction on the typing derivation; it does not

require much extra machinery over and above the reduction relation. In contrast, the proof of soundness in Brunel et al. [2014] requires a realizability model on top of the reduction relation. In this regard, this soundness theorem is more like the modality preservation theorem in Abel and Bernardy [2020]. However, the modality preservation theorem in Abel and Bernardy [2020] is not employed to reason about no use and linear use, unlike our soundness theorem which can be readily employed to reason about such uses.

## 4.7 Discussion

### 4.7.1 Technical Comparison with QTT

As discussed in Chapter 1, GrAD is closely related to QTT. However, the usage accounting principles in the two calculi differ. Here, we look at the technical implications of this difference between GrAD and QTT. Recall that QTT tracks usage in types and terms differently: types live in a resource-agnostic world while terms live in a resource-aware world. So QTT employs two forms of typing judgments: a resource-agnostic form, $\Gamma \vdash A :^0 \texttt{Set}$, for types and a resource-aware form, $\Gamma \vdash a :^1 A$, for terms. Here, $\Gamma$ is a context graded by elements of a semiring, $A$ is a type and $a$ is a term. On the other hand, GrAD employs only one form of typing judgment (for both types and terms) that is resource-aware. As we pointed out earlier, a single form of typing judgment results in a simpler and more uniform calculus. It also leads to less restrictions on the parametrizing structure. To elaborate, QTT requires the parametrizing semiring to be zerosumfree and entire (see Section 4.4.2) to admit substitution. In contrast, GrAD requires no such restrictions for admissibility of substitution. However, there is a benefit in having a resource-agnostic fragment in QTT. Unlike GrAD, QTT can support strong $\Sigma$-types via its resource-agnostic fragment. In Chapter 5, we shall design a calculus that incorporates the best of both QTT and GrAD into a single system. That calculus would treat types and terms uniformly and would also have a resource-agnostic fragment that supports strong $\Sigma$-types.

Before closing this comparison, we shall point out a small but significant technical difference between GrAD and QTT: GrAD includes sub-usaging, while QTT does not. Note here that since QTT is parametrized by a semiring without an order, a sub-usaging rule does not make sense in the calculus. But the absence of sub-usaging has important consequences. In particular, QTT can only track exact usage. It cannot track bounded usage. Further, the calculus cannot allow linear use of unrestricted resources, i.e. the judgment, $x :^\omega A \vdash x :^1 A$, does not hold in QTT, parametrized over the semiring $\{0, 1\omega\}$.

### 4.7.2 Heap Semantics for Linear Calculi

Computational and operational interpretations of linear logic have been explored in several works, especially in Chirimar et al. [2000], Turner and Wadler [1999]. Turner and Wadler [1999]

provide a heap-based operational interpretation of linear logic. They show that a call-by-name linear calculus enjoys the single pointer property, i.e. a linear resource has exactly one pointer to it while a call-by-need linear calculus satisfies a weaker version of this property, guaranteeing only the maintenance of a single pointer. Their system considers only linear and unrestricted resources. We generalize their operational interpretation of linear logic to a graded dependent call-by-name calculus where resources can be drawn from an arbitrary preordered semiring. We also generalize the single pointer property to our graded setting. However, unlike Turner and Wadler [1999], we don't explore a call-by-need semantics for our calculus.

## 4.8 Conclusion

Graded type systems based on BLL are a generic framework for analyzing usage of resources in programs. This chapter provides a new way of incorporating this framework into dependently-typed languages, with the goal of supporting both no use and linear use in the same system. We designed a graded dependent type system, GraD, and presented a standard substitution-based semantics and a usage-aware heap-based semantics for the calculus. Heap-based semantics, unlike substitution-based semantics, can track usage and enforce fairness constraints on usage. We showed that GraD is sound with respect to a heap-based semantics that ensures fair use. From this result, we conclude that GraD analyzes usage correctly.

# Chapter 5

# Combined Linearity and Dependency Analysis in Pure Type Systems

In Chapter 3, we analyzed dependencies in pure type systems. In Chapter 4, we analyzed linearity in pure type systems. In this chapter, we unify these analyses. There are several benefits to a unification of linearity and dependency analyses. First, from a theoretical perspective, it would unify the standard calculi for linearity analysis [Barber, 1996, Benton, 1994] with the standard calculi for dependency analysis [Abadi et al., 1999, Shikuma and Igarashi, 2006]. Second, from a practical perspective, it would allow programmers to use the same type system for both linearity and dependency analyses. Presently, programmers use different type systems for this purpose. For example, Haskell programmers use Linear Haskell library [Bernardy et al., 2018] for linearity analysis and LIO library [Stefan et al., 2017] for dependency analysis. Third, it would allow a combination of the two analyses. A combined analysis is more powerful than the individual analyses done separately because it allows arbitrary combination of usage and flow constraints. For example, in a combined analysis, a piece of data may be simultaneously linear and secure.

With recent advances in graded type systems [Abel and Scherer, 2012, Brunel et al., 2014, Ghica and Smith, 2014, Orchard et al., 2019, Petricek et al., 2014], it has been realized that the two analyses can be viewed through the same lens. However, though existing graded type systems analyze linearity well, they are limited in their dependency analysis. There are several aspects of dependency analysis that these systems cannot capture. We discuss them in detail in the next section. The main reason behind their shortcoming is that they have been designed for analyzing coeffects, i.e. how programs depend upon their contexts. Coeffects include linearity (single use), irrelevance (no use), etc. Dependency, however, behaves more like an effect. To elaborate: low and high security computations may be seen as pure and effectful computations respectively. An

effect like dependency is not well-captured by existing graded type systems which are basically coeffect calculi.

In this chapter, we show that by systematically extending existing graded type systems, we can use them for a general linearity and dependency analysis. We design a calculus, LDC, that can simultaneously analyze, using the same mechanism, a coeffect like linearity and an effect like dependency. LDC is parametrized by an arbitrary Pure Type System and it subsumes standard calculi for linearity and dependency analyses. We show that linearity and dependency analyses in LDC are correct using a heap semantics.

In summary, we make the following contributions in this chapter:

- We present a language, LDC, parametrized by an arbitrary pure type system, that analyzes linearity and dependency using the same mechanism.
- We show that LDC subsumes the standard calculi for analyzing linearity and dependency like Linear Nonlinear $\lambda$-calculus of Benton [1994], DCC of Abadi et al. [1999], Sealing Calculus of Shikuma and Igarashi [2006], etc.
- We show that correctness of both linearity and dependency analyses in LDC follow from the soundness theorem for the calculus.
- We show that LDC can carry out a combined linearity and dependency analysis.

## 5.1 Challenges and Resolution

### 5.1.1 Dependency Analysis: Salient Aspects

Recall that dependency type systems [Abadi et al., 1999, Shikuma and Igarashi, 2006] are based on the lattice model of information flow. These type systems grade the monadic modality, $T$, of Moggi's computational metalanguage with labels drawn from a dependency lattice. To enable dependency analysis, this modality needs to support a graded join function (see Chapter 1). If dependency labels are seen as effects, then the join function corresponds to computing the union of these effects. Just as computing union is important in an effect calculus, having a join operation is important in a dependency calculus. Later in this section, we shall see that existing graded type systems cannot derive a join operation. This significantly limits dependency analysis in these systems.

Before moving further, we want to point out another important aspect of dependency analysis that is sometimes ignored. It pertains to the treatment of functions that are wrapped under $T_\ell$'s. Consider the type: $T_{\mathbf{H}}\,(T_{\mathbf{H}}\,\mathbf{Bool} \to \mathbf{Bool})$. What should the values of this type be? DCC [Abadi et al., 1999] would answer: just $\eta_{\mathbf{H}}\,(\lambda x.\mathbf{true})$ and $\eta_{\mathbf{H}}\,(\lambda x.\mathbf{false})$. But Sealing Calculus [Shikuma and Igarashi, 2006] would answer: $[\lambda x.\mathbf{true}]_{\mathbf{H}}$, $[\lambda x.\mathbf{false}]_{\mathbf{H}}$, $[\lambda x.x^{\mathbf{H}}]_{\mathbf{H}}$ and $[\lambda x.\mathbf{not}\ (x^{\mathbf{H}})]_{\mathbf{H}}$.

This difference stems from the fact that in Sealing Calculus, the function $T_{\mathbf{H}}\,\mathbf{Bool} \to \mathbf{Bool}$, if wrapped under $T_{\mathbf{H}}$, may return a high-security output, unlike DCC where it must always be constant. In this regard, Sealing Calculus is more general than DCC because it doesn't restrict any function from returning high-security values, if the function itself is wrapped under $T_{\mathbf{H}}$. Note here that over terminating computations, Sealing Calculus subsumes DCC and is, in fact, more general than DCC, as we see above. So we use the Sealing Calculus as our model of dependency analysis. As an aside, we allow nonterminating computations in our language even though Sealing Calculus does not.

### 5.1.2 Graded Type Systems: Salient Aspects

Over the recent years, graded type systems [Abel and Scherer, 2012, Atkey, 2018, Brunel et al., 2014, Choudhury et al., 2021, Gaboardi et al., 2016, Ghica and Smith, 2014, McBride, 2016, Moon et al., 2021, Orchard et al., 2019, Petricek et al., 2014] have been successfully employed for quantitative reasoning in programs. As we pointed out earlier, these systems can carry out fine-grained quantitative analysis. The power and flexibility of these systems stem from the fact that they are parametrized by an abstract preordered semiring ($\mathcal{Q}$) or a similar structure that represents an algebra of resources. By varying $\mathcal{Q}$, these systems can carry out a variety of analyses.

Recall that graded type systems use the elements of the parametrizing preordered semiring to grade the !-modality. Now, quantitative analysis requires splitting of resources through functions like $!_{q_1 \cdot q_2}\, A \to !_{q_1}\, !_{q_2}\, A$, for an arbitrary type $A$ and $q_1, q_2 \in \mathcal{Q}$. The !-modality, being comonadic, supports this function through the standard graded fork function (see Chapter 1). However, since quantitative analysis does not require the inverse function, these systems do not derive a monadic join. As a matter of fact, these systems cannot derive a monadic join in general. Choudhury [2022c] gives a model-theoretic argument showing why.

### 5.1.3 Limitations of Dependency Analysis in Graded Type Systems

Next, we point out the limitations of dependency analysis in existing graded type systems. First, as discussed above, these type systems are parametrized by preordered semirings. Dependency analysis, however, is parametrized by lattices. Recall from Chapter 1 that an arbitrary lattice cannot be viewed as a preordered semiring. Thus, these systems cannot analyze dependencies over arbitrary lattices.

Second, as discussed above, dependency analysis needs a join operator. However, existing graded type systems cannot derive a join operator. An ad hoc solution to this problem might be to add an explicit join operator to the type system, for example, an operator **join** of type $!_{q_1}\, !_{q_2}\, A \to !_{q_1 \cdot q_2}\, A$ for any $A$ and $q_1, q_2 \in \mathcal{Q}$. However, such a solution creates several issues. First, what should the semantics of this operator be? If $v$ is a value, should $\mathbf{join}\ v$ be also a value? Then, we would

have more values like $\mathbf{join}\,!_{q_1}\,!_{q_2}\,\mathbf{true}$. Alternatively, if $\mathbf{join}\,v$ is not a value, how should it step? One might answer $\mathbf{join}\,!_{q_1}\,!_{q_2}\,v$ should step to $!_{q_1 \cdot q_2}\,v$. While such a stepping rule would make sense in a call-by-value calculus, it would not in a call-by-name calculus because there $!_q\,a$ is a value, irrespective of whether $a$ itself is a value or not. Second, one would expect **join** to be the inverse of fork that is already derivable in these systems. However, since **join** is added as an ad hoc construct, there is no principled way to ensure this property. Third, the addition of this construct breaks the symmetry of the type system since the corresponding typing rule is neither an introduction rule nor an elimination rule for !. So we avoid this solution towards enabling dependency analysis in graded type systems.

Third, as discussed above, we want a dependency calculus where any function can return high-security values, whenever the function itself is wrapped under a high-security label. Existing graded type systems do not allow this. For a lattice, $\mathbf{L} \sqsubseteq \mathbf{H}$, the type $!_{\mathbf{H}}\,(!_{\mathbf{H}}\,\mathbf{Bool} \to \mathbf{Bool})$ contains only two distinct terms in these systems: $!_{\mathbf{H}}\,(\lambda x.\mathbf{true})$ and $!_{\mathbf{H}}\,(\lambda x.\mathbf{false})$.

Hence, we see that there are several limitations of dependency analysis in existing graded type systems. This motivates us to look for other solutions for unifying linearity and dependency analyses.

### 5.1.4 Towards Resolution

Our dependency calculus from Chapter 3, DDC$^\top$, points towards a solution. Though similar to existing graded type systems, DDC$^\top$ avoids all their limitations listed above. It subsumes the Sealing Calculus and it can analyze dependencies in a dependent setting. The key difference between DDC$^\top$ and the graded type systems discussed above is in the form of their typing judgment. Graded type systems use a typing judgment of the form:

$$x_1 :^{q_1} A_1, x_2 :^{q_2} A_2, \ldots, x_n :^{q_n} A_n \vdash b : B,$$

where $q_i \in \mathcal{Q}$. On the other hand, DDC$^\top$ uses a typing judgment of the form:

$$x_1 :^{\ell_1} A_1, x_2 :^{\ell_2} A_2, \ldots, x_n :^{\ell_n} A_n \vdash b :^{\ell} B,$$

where $\ell_i, \ell \in \mathcal{L}$. Note the extra label to the right of the turnstile! The label to the right symbolizes the observer's world in DDC$^\top$. In graded type systems, the observer's world is fixed at 1. This added flexibility enables DDC$^\top$ carry out a general dependency analysis.

However, DDC$^\top$ cannot carry out linearity or for that matter, any quantitative analysis. So the problem of unifying linearity and dependency analyses still remains open. One possible solution might be to allow graded type systems to vary the observer's world using typing judgments of the form:

$$x_1 :^{q_1} A_1, x_2 :^{q_2} A_2, \ldots, x_n :^{q_n} A_n \vdash b :^{q} B \tag{5.1}$$

where $q_i, q \in \mathcal{Q}$. There is, however, a known roadblock that awaits us along this direction. Atkey [2018] showed that a type system that uses the above judgment form and is parametrized by an arbitrary semiring does not admit substitution. But all is not lost! We find that though substitution is inadmissible over some semirings, it is in fact admissible over several preordered semirings that interest us. In particular, substitution is admissible over the standard preordered semirings used for tracking linear and affine use. Recall that both these semirings, denoted $\mathcal{Q}_{\text{Lin}}$ and $\mathcal{Q}_{\text{Aff}}$ respectively, have 3 elements: $0, 1$ and $\omega$, with $1 + 1 = \omega + 1 = 1 + \omega = \omega + \omega = \omega$ and $\omega \cdot \omega = \omega$. However, $\mathcal{Q}_{\text{Lin}}$ is ordered as: $\omega <: 0$ and $\omega <: 1$ while $\mathcal{Q}_{\text{Aff}}$ is ordered as: $\omega <: 1 <: 0$.

Thus, we finally have a way to unify linearity and dependency analyses. In our Linearity Dependency Calculus, LDC, we use typing judgments of the form shown in (5.1). For linearity and other quantitative analyses, we parametrize LDC over certain preordered semirings that we describe as we go along. For dependency analysis, we parametrize LDC over an *arbitrary* lattice.

Now, LDC can analyze linearity and dependency in an arbitrary pure type system. However, some of the key ideas of the calculus are best explained in a simply-typed setting. So, we first present the simply-typed version of LDC and thereafter generalize it to its pure type system version.

## 5.2 Linearity and Dependency Analyses over Simple Types

### 5.2.1 Type System for Linearity Analysis

First, we analyze exact use and bounded use. We shall add unrestricted use, referred to by $\omega$, to our calculus in Section 5.5. Exact use can be analyzed by $\mathbb{N}_{=}$, the semiring of natural numbers with discrete order. Bounded use can be analyzed by $\mathbb{N}_{\geq}$, the semiring of natural numbers with descending natural order. The ordering in $\mathbb{N}_{\geq}$ looks like: $\ldots <: 4 <: 3 <: 2 <: 1 <: 0$. The reason behind the difference in ordering is that in bounded use analysis, resources may be discarded. But note that resources are never copied in either of these analyses.

Next, we present LDC parametrized over these semirings. Whenever we need precision, we refer to LDC parametrized over an algebraic structure, $\mathcal{AS}$, as LDC($\mathcal{AS}$). The algebraic structure, $\mathcal{AS}$, may be either a preordered semiring or a lattice or in the case of combined analysis, their cartesian product.

Now, let $\mathcal{Q}_{\mathbb{N}}$ vary over $\{\mathbb{N}_{=}, \mathbb{N}_{\geq}\}$. The grammar of LDC($\mathcal{Q}_{\mathbb{N}}$) appears in Figure 5.1. It is similar to that of the BLL-based simple type system from Chapter 4: there are function types, weak and strong product types, sum types and a **Unit** type along with their introduction and elimination forms. Like the BLL-based system, LDC uses graded contexts. However, LDC has a more general typing judgment. The typing judgment of LDC, $\Gamma \vdash a :^{q} A$, may be intuitively understood as: the resources in $\Gamma$ can produce $q$ copies of $A$. With this understanding, let us look at the type

$$
\begin{array}{ll}
\text{types}, A, B, C & ::= \mathbf{Unit} \mid {}^{r}A \to B \mid {}^{r}A_1 \times A_2 \mid A_1 \,\&\, A_2 \mid A_1 + A_2 \\
\text{terms}, a, b, c & ::= x \mid \lambda^{r} x : A.b \mid b\ a^{r} \\
& \mid \mathbf{unit} \mid \mathbf{let}_{q_0}\ \mathbf{unit} = a\ \mathbf{in}\ b \mid ({a_1}^{r}, a_2) \mid \mathbf{let}_{q_0}\ (x^{r}, y) = a\ \mathbf{in}\ b \\
& \mid (a_1; a_2) \mid \mathbf{proj}_1\ a \mid \mathbf{proj}_2\ a \mid \mathbf{inj}_1\ a_1 \mid \mathbf{inj}_2\ a_2 \mid \mathbf{case}_{q_0}\ a\ \mathbf{of}\ x_1.b_1\ ;\ x_2.b_2 \\
\text{contexts}, \Gamma & ::= \varnothing \mid \Gamma, x :^{q} A
\end{array}
$$

FIGURE 5.1: Grammar of LDC($\mathcal{Q}_{\mathbb{N}}$) (Simple Version)

$\boxed{\Gamma \vdash a :^{q} A}$ *(Simple Version)*

$$\text{ST-Var}\ \frac{}{0 \cdot \Gamma_1, x :^{q} A, 0 \cdot \Gamma_2 \vdash x :^{q} A} \qquad \text{ST-Lam}\ \frac{\Gamma, x :^{q \cdot r} A \vdash b :^{q} B}{\Gamma \vdash \lambda^{r} x : A.b :^{q}\ {}^{r}A \to B} \qquad \text{ST-App}\ \frac{\Gamma_1 \vdash b :^{q}\ {}^{r}A \to B \quad \Gamma_2 \vdash a :^{q \cdot r} A \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash b\ a^{r} :^{q} B}$$

$$\text{ST-WPair}\ \frac{\Gamma_1 \vdash a_1 :^{q \cdot r} A_1 \quad \Gamma_2 \vdash a_2 :^{q} A_2 \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash ({a_1}^{r}, a_2) :^{q}\ {}^{r}A_1 \times A_2} \qquad \text{ST-LetPair}\ \frac{\Gamma_1 \vdash a :^{q \cdot q_0}\ {}^{r}A_1 \times A_2 \quad \Gamma_2, x :^{q \cdot q_0 \cdot r} A_1, y :^{q \cdot q_0} A_2 \vdash b :^{q} B \quad q_0 <: 1 \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}_{q_0}\ (x^{r}, y) = a\ \mathbf{in}\ b :^{q} B}$$

$$\text{ST-Unit}\ \frac{}{0 \cdot \Gamma \vdash \mathbf{unit} :^{q} \mathbf{Unit}} \qquad \text{ST-LetUnit}\ \frac{\Gamma_1 \vdash a :^{q \cdot q_0} \mathbf{Unit} \quad \Gamma_2 \vdash b :^{q} B \quad q_0 <: 1 \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}_{q_0}\ \mathbf{unit} = a\ \mathbf{in}\ b :^{q} B} \qquad \text{ST-SPair}\ \frac{\Gamma \vdash a_1 :^{q} A_1 \quad \Gamma \vdash a_2 :^{q} A_2}{\Gamma \vdash (a_1; a_2) :^{q} A_1 \,\&\, A_2}$$

$$\text{ST-Proj1}\ \frac{\Gamma \vdash a :^{q} A_1 \,\&\, A_2}{\Gamma \vdash \mathbf{proj}_1\ a :^{q} A_1} \qquad \text{ST-Proj2}\ \frac{\Gamma \vdash a :^{q} A_1 \,\&\, A_2}{\Gamma \vdash \mathbf{proj}_2\ a :^{q} A_2} \qquad \text{ST-Inj1}\ \frac{\Gamma \vdash a_1 :^{q} A_1}{\Gamma \vdash \mathbf{inj}_1\ a_1 :^{q} A_1 + A_2} \qquad \text{ST-Inj2}\ \frac{\Gamma \vdash a_2 :^{q} A_2}{\Gamma \vdash \mathbf{inj}_2\ a_2 :^{q} A_1 + A_2}$$

$$\text{ST-Case}\ \frac{\Gamma_1 \vdash a :^{q \cdot q_0} A_1 + A_2 \quad \Gamma_2, x_1 :^{q \cdot q_0} A_1 \vdash b_1 :^{q} B \quad \Gamma_2, x_2 :^{q \cdot q_0} A_2 \vdash b_2 :^{q} B \quad q_0 <: 1 \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{q_0}\ a\ \mathbf{of}\ x_1.b_1\ ;\ x_2.b_2 :^{q} B}$$

$$\text{ST-SubL}\ \frac{\Gamma' \vdash a :^{q} A \qquad \Gamma <: \Gamma'}{\Gamma \vdash a :^{q} A} \qquad \text{ST-SubR}\ \frac{\Gamma \vdash a :^{q} A \qquad q <: q'}{\Gamma \vdash a :^{q'} A}$$

FIGURE 5.2: Typing rules for LDC($\mathcal{Q}_{\mathbb{N}}$)

system that appears in Figure 5.2. Note here that the operations and relation on contexts are defined in the same way as in Chapter 4.

Most of the typing rules are as expected. A point to note is that the elimination rules ST-LetPair, ST-LetUnit, and ST-Case have a side condition $q_0 <: 1$. The reason behind this condition is that for reducing any of these elimination forms, we first need to reduce the term $a$, implying that in any case, we need the resources for reducing at least one copy of $a$. Further,

the grade $q_0$ makes these rules more flexible. For example, owing to this flexibility, in rule ST-LetPair, $b$ may use $q \cdot q_0$ copies of $y$ in lieu of just $q$ copies of $y$. Another point to note is how the rules ST-SubL and ST-SubR help discard resources in LDC($\mathbb{N}_{\geq}$).

Now, let us look at a few examples of derivable and non-derivable terms in LDC($\mathcal{Q}_{\mathbb{N}}$). (To reduce complexity, we omit the domain types in the $\lambda$s.)

$$\varnothing \vdash \lambda^1 x.x :^1 {}^1A \to A \text{ but } \varnothing \nvdash \lambda^0 x.x :^1 {}^0A \to A$$
$$\varnothing \vdash \lambda^1 x.x :^2 {}^1A \to A \text{ but } \varnothing \nvdash \lambda^1 x.(x^2, \mathbf{unit}) :^1 {}^1A \to {}^2A \times \mathbf{Unit}$$
$$\varnothing \vdash \lambda^1 x.(x; x) :^1 {}^1A \to A \,\&\, A \text{ but } \varnothing \nvdash \lambda^1 x.(x^1, x) :^1 {}^1A \to {}^1A \times A$$

On a closer look, we find that the non-derivable terms above can not use resources fairly. Consider the types of these terms: ${}^0A \to A$, ${}^1A \to {}^2A \times \mathbf{Unit}$ and ${}^1A \to {}^1A \times A$. Such types may be inhabited only if resources can be copied. Since we disallow copying, they are essentially uninhabited.

Note that in order to produce 0 copy of any term, we do not need any resource. So in the 0-world, any annotated type is inhabited, provided its unannotated counterpart is inhabited. However, resources do not have any meaning in the 0-world. In other words, the judgment $\Gamma \vdash a :^0 A$ conveys no more information than its corresponding standard $\lambda$-calculus counterpart.

Next, we look at the operational semantics and metatheory of LDC($\mathcal{Q}_{\mathbb{N}}$).

### 5.2.2 Metatheory of Linearity Analysis

First, we consider some syntactic properties. The calculus satisfies the multiplication lemma stated below. This lemma says that if we need $\Gamma$ resources to produce $q$ copies of $a$, then we would need $r_0 \cdot \Gamma$ resources to produce $r_0 \cdot q$ copies of $a$.

**Lemma 5.1** (Multiplication) If $\Gamma \vdash a :^q A$, then $r_0 \cdot \Gamma \vdash a :^{r_0 \cdot q} A$.

The calculus also satisfies a factorization lemma, stated below. This lemma says that if a context $\Gamma$ produces $q$ copies of $a$, then $\Gamma$ can be split into $q$ parts whereby each part produces 1 copy of $a$. We need the precondition $q \neq 0$ since resources don't have any meaning in the 0-world but they are meaningful in all other worlds, in particular, the 1-world.

**Lemma 5.2** (Factorization) If $\Gamma \vdash a :^q A$ and $q \neq 0$, then there exists $\Gamma'$ such that $\Gamma' \vdash a :^1 A$ and $\Gamma <: q \cdot \Gamma'$.

Using the above two lemmas, we can prove a splitting lemma stated below. This lemma says that if we have the resources, $\Gamma$, to produce $q_1 + q_2$ copies of $a$, then we can split $\Gamma$ into two parts, $\Gamma_1$ and $\Gamma_2$, such that $\Gamma_1$ and $\Gamma_2$ can produce $q_1$ and $q_2$ copies of $a$ respectively. Atkey [2018] showed that in a system similar to ours, the splitting lemma does not hold for some semirings. However, we find that it holds for the preordered semirings parametrizing LDC, for example, any $\mathcal{Q}_{\mathbb{N}}$.

$\boxed{\vdash a \rightsquigarrow a'}$ *(Excerpt)*

$$\textsc{Step-AppBeta}\ \frac{}{\vdash (\lambda^r x : A.b)\ a^r \rightsquigarrow b\{a/x\}} \qquad \textsc{Step-LetPairBeta}\ \frac{}{\vdash \mathbf{let}_{q_0}\ (x^r, y) = ({a_1}^r, a_2)\ \mathbf{in}\ b \rightsquigarrow b\{a_1/x\}\{a_2/y\}}$$

FIGURE 5.3: Small-step reduction for LDC($\mathcal{Q}_{\mathbb{N}}$)

**Lemma 5.3** (Splitting) If $\Gamma \vdash a :^{q_1+q_2} A$, then there exists $\Gamma_1$ and $\Gamma_2$ such that $\Gamma_1 \vdash a :^{q_1} A$ and $\Gamma_2 \vdash a :^{q_2} A$ and $\Gamma = \Gamma_1 + \Gamma_2$.

Next, we look at weakening and substitution. For substitution, the allowed usage of the variable should match the number of available copies of the substitute. Further, the term, after substitution, would need the combined resources.

**Lemma 5.4** (Weakening) If $\Gamma_1, \Gamma_2 \vdash a :^q A$, then $\Gamma_1, z :^0 C, \Gamma_2 \vdash a :^q A$.

**Lemma 5.5** (Substitution) If $\Gamma_1, z :^{r_0} C, \Gamma_2 \vdash a :^q A$ and $\Gamma \vdash c :^{r_0} C$ and $\lfloor \Gamma_1 \rfloor = \lfloor \Gamma \rfloor$, then $\Gamma_1 + \Gamma, \Gamma_2 \vdash a\{c/z\} :^q A$.

Now, we consider the operational semantics of the language. Our operational semantics is a call-by-name small-step semantics. All the step rules are standard other than the two rules that appear in Figure 5.3. With this operational semantics, our language enjoys the standard type soundness property.

**Theorem 5.6** (Preservation) If $\Gamma \vdash a :^q A$ and $\vdash a \rightsquigarrow a'$, then $\Gamma \vdash a' :^q A$.

**Theorem 5.7** (Progress) If $\varnothing \vdash a :^q A$, then either $a$ is a value or there exists $a'$ such that $\vdash a \rightsquigarrow a'$.

We know that standard substitution-based semantics cannot really model resource usage of programs. So, in Section 5.3, we extend our heap-based semantics from Chapter 4 to this calculus and show that the type system accounts usage correctly. But for now, we move on to dependency analysis.

### 5.2.3 Type System for Dependency Analysis

Let $\mathcal{L} = (L, \sqcap, \sqcup, \top, \bot, \sqsubseteq)$ be an arbitrary lattice. We use $\ell, m$ to denote elements of $L$. Now, interpreting $+, \cdot, 0, 1$ and $<:$ as $\sqcap, \sqcup, \top, \bot$ and $\sqsubseteq$ respectively, and using $q$ and $r$ for the elements of $\mathcal{L}$, we have a dependency calculus in the type system presented in Figure 5.2. In Figure 5.4, we present a few selected rules from this type system with just changed notation.

The type system is now parametrized by $\mathcal{L}$ in lieu of $\mathcal{Q}_{\mathbb{N}}$. We define $\ell \sqcup \Gamma$, $\Gamma_1 \sqcap \Gamma_2$ and $\Gamma \sqsubseteq \Gamma'$ in the same way as their counterparts $q \cdot \Gamma$, $\Gamma_1 + \Gamma_2$ and $\Gamma <: \Gamma'$ respectively. The typing judgment $x_1 :^{\ell_1} A_1, x_2 :^{\ell_2} A_2, \ldots, x_n :^{\ell_n} A_n \vdash b :^{\ell} B$ may be read as: $b$ is visible at $\ell$, assuming $x_i$ is visible at $\ell_i$ for $i = 1, 2, \ldots, n$. With this reading, let us look at the type system in Figure 5.4. The first

$\boxed{\Gamma \vdash a :^{\ell} A}$ *(Selected rules)*

$$\text{ST-VarD}\ \frac{}{\top \sqcup \Gamma_1, x :^{\ell} A, \top \sqcup \Gamma_2 \vdash x :^{\ell} A} \qquad \text{ST-LamD}\ \frac{\Gamma, x :^{\ell \sqcup m} A \vdash b :^{\ell} B}{\Gamma \vdash \lambda^m x : A.b :^{\ell}\ ^m A \to B} \qquad \text{ST-AppD}\ \frac{\Gamma_1 \vdash b :^{\ell}\ ^m A \to B \quad \Gamma_2 \vdash a :^{\ell \sqcup m} A \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 \sqcap \Gamma_2 \vdash b\ a^m :^{\ell} B}$$

$$\text{ST-WPairD}\ \frac{\Gamma_1 \vdash a_1 :^{\ell \sqcup m} A_1 \quad \Gamma_2 \vdash a_2 :^{\ell} A_2 \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 \sqcap \Gamma_2 \vdash ({a_1}^m, a_2) :^{\ell}\ ^m A_1 \times A_2}$$

$$\text{ST-LetPairD}\ \frac{\Gamma_1 \vdash a :^{\ell \sqcup \ell_0}\ ^m A_1 \times A_2 \quad \Gamma_2, x :^{\ell \sqcup \ell_0 \sqcup m} A_1, y :^{\ell \sqcup \ell_0} A_2 \vdash b :^{\ell} B \quad \ell_0 \sqsubseteq \bot \quad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 \sqcap \Gamma_2 \vdash \mathbf{let}_{\ell_0}\ (x^m, y) = a\ \mathbf{in}\ b :^{\ell} B}$$

$$\text{ST-SubLD}\ \frac{\Gamma' \vdash a :^{\ell} A \qquad \Gamma \sqsubseteq \Gamma'}{\Gamma \vdash a :^{\ell} A} \qquad \text{ST-SubRD}\ \frac{\Gamma \vdash a :^{\ell} A \qquad \ell \sqsubseteq \ell'}{\Gamma \vdash a :^{\ell'} A}$$

FIGURE 5.4: Typing rules of LDC($\mathcal{L}$)

thing we notice is the similarity of this type system with that of SDC, presented in Chapter 3. Most of the typing rules are just variants of the ones for SDC. However, there are some minor differences.

One technical difference is that SDC does not have security annotations on function and product types. SDC makes up for it through its modal types, $T_\ell\ A$. Annotated function and product types of LDC($\mathcal{L}$), ${}^m A \to B$ and ${}^m A \times B$, correspond to types $T_m\ A \to B$ and $T_m\ A \times B$ respectively in SDC. Conversely, the modal type of SDC, $T_m\ A$, corresponds to ${}^m A \times \mathbf{Unit}$ in LDC($\mathcal{L}$).

Another difference is that LDC($\mathcal{L}$) has explicit typing rules for relaxing the security constraints on the context (rule ST-SubLD) and for strengthening the security constraint on the observer (rule ST-SubRD). SDC, on the other hand, allows these operations implicitly via its variable rule.

Also, note that rule ST-LetPairD has a side-condition, $\ell_0 \sqsubseteq \bot$. Since $\bot$ is the bottom element, this side-condition forces $\ell_0$ to be $\bot$. So rule ST-LetPairD may be simplified with $\ell_0$ set to $\bot$. However, we present it in this way to emphasize the similarity between LDC($\mathcal{Q}_{\mathbb{N}}$) and LDC($\mathcal{L}$).

Next, we consider a few examples of derivable and non-derivable terms in LDC($\mathcal{L}_\diamond$), where $\mathcal{L}_\diamond$ is the diamond lattice (see Chapter 1).

Let $\mathbf{Bool} \coloneqq \mathbf{Unit} + \mathbf{Unit}$ and let $\mathbf{true} \coloneqq \mathbf{inj}_1\ \mathbf{unit}$ and $\mathbf{false} \coloneqq \mathbf{inj}_2\ \mathbf{unit}$. Then,

$$\varnothing \vdash \lambda^{\mathbf{L}} x.x :^{\mathbf{H}\ \mathbf{L}} \mathbf{Bool} \to \mathbf{Bool} \text{ but } \varnothing \nvdash \lambda^{\mathbf{H}} x.x :^{\mathbf{L}\ \mathbf{H}} \mathbf{Bool} \to \mathbf{Bool}$$

$$\varnothing \vdash \lambda^{\mathbf{H}} x.\eta_{\mathbf{M_1}}\ \eta_{\mathbf{M_2}}\ x :^{\mathbf{L}\ \mathbf{H}} \mathbf{Bool} \to T_{\mathbf{M_1}}\ T_{\mathbf{M_2}}\ \mathbf{Bool} \text{ but } x :^{\mathbf{H}}\ \mathbf{Bool} \nvdash x :^{\mathbf{M_1}}\ \mathbf{Bool}$$

Here, $T_m\ A \triangleq {}^mA \times \mathbf{Unit}$ and $\eta_\ell\ a \triangleq (a^\ell, \mathbf{unit})$. On a closer look, we find that the non-derivable terms violate the dependence structure of $\mathcal{L}_\diamond$. The first term transfers information from $\mathbf{H}$ to $\mathbf{L}$ while the second one does so from $\mathbf{H}$ to $\mathbf{M_1}$. The derivable terms, on the other hand, respect the dependence structure of $\mathcal{L}_\diamond$. The first term transfers information from $\mathbf{L}$ to $\mathbf{H}$ while the second one emboxes $\mathbf{H}$ information in an $\mathbf{M_2}$ box nested within an $\mathbf{M_1}$ box.

Now, we present the terms that witness the standard join and fork operations in LDC($\mathcal{L}$). (Note that any erased annotation is assumed to be $\bot$.)

$$\varnothing \vdash c_1 : T_{\ell_1}\ T_{\ell_2}\ A \to T_{\ell_1 \sqcup \ell_2}\ A \text{ and } \varnothing \vdash c_2 : T_{\ell_1 \sqcup \ell_2}\ A \to T_{\ell_1}\ T_{\ell_2}\ A \text{ where,}$$
$$c_1 \coloneqq \lambda x.\eta_{\ell_1 \sqcup \ell_2}\ (\mathbf{let}\ (y^{\ell_2}, \_) = (\mathbf{let}\ (z^{\ell_1}, \_) = x\ \mathbf{in}\ z)\ \mathbf{in}\ y)$$
$$c_2 \coloneqq \lambda x.\eta_{\ell_1}\ \eta_{\ell_2}\ (\mathbf{let}\ (y^{\ell_1 \sqcup \ell_2}, \_) = x\ \mathbf{in}\ y)$$

Next, we look at the metatheory of LDC($\mathcal{L}$).

### 5.2.4 Metatheory of Dependency Analysis

We consider the dependency counterparts of the properties of LDC($\mathcal{Q}_\mathbb{N}$), presented in Section 5.2.2. Most of these properties are true of LDC($\mathcal{L}$). LDC($\mathcal{L}$) satisfies the multiplication lemma. The lemma says that we can always simultaneously upgrade the context and the level at which the derived term is viewed.

**Lemma 5.8** (Multiplication) If $\Gamma \vdash a :^\ell A$, then $m_0 \sqcup \Gamma \vdash a :^{m_0 \sqcup \ell} A$.

The splitting lemma is also true. Since $\sqcap$ is idempotent, it follows directly from rule ST-SubRD.

**Lemma 5.9** (Splitting) If $\Gamma \vdash a :^{q_1 \sqcap q_2} A$, then there exists $\Gamma_1$ and $\Gamma_2$ such that $\Gamma_1 \vdash a :^{q_1} A$ and $\Gamma_2 \vdash a :^{q_2} A$ and $\Gamma = \Gamma_1 \sqcap \Gamma_2$.

The factorization lemma, however, is not true here because if true, it would allow information to leak. To see how, consider the following LDC($\mathcal{L}_\diamond$) judgment: $\varnothing \vdash \lambda x.\mathbf{let}\ (y^{\mathbf{M_1}}, \_) = x\ \mathbf{in}\ y :^{\mathbf{M_1}} T_{\mathbf{M_1}}\ \mathbf{Bool} \to \mathbf{Bool}$. If the factorization lemma were true, this judgment would give us: $\varnothing \vdash \lambda x.\mathbf{let}\ (y^{\mathbf{M_1}}, \_) = x\ \mathbf{in}\ y :^{\mathbf{L}} T_{\mathbf{M_1}}\ \mathbf{Bool} \to \mathbf{Bool}$, a non-constant function from $T_{\mathbf{M_1}}\ \mathbf{Bool}$ to $\mathbf{Bool}$ in an $\mathbf{L}$-secure world, representing a leak of information. Note that in LDC($\mathcal{Q}_\mathbb{N}$), the factorization lemma is required for proving the splitting lemma which, in turn, is necessary to show substitution. In LDC($\mathcal{L}$), the splitting lemma holds trivially and does not need any factorization lemma.

Next, we consider weakening and substitution. The substitution lemma substitutes an assumption held at $m_0$ with a term derived at $m_0$.

**Lemma 5.10** (Weakening) If $\Gamma_1, \Gamma_2 \vdash a :^\ell A$, then $\Gamma_1, z :^\top C, \Gamma_2 \vdash a :^\ell A$.

**Lemma 5.11** (Substitution) If $\Gamma_1, z :^{m_0} C, \Gamma_2 \vdash a :^\ell A$ and $\Gamma \vdash c :^{m_0} C$ and $\lfloor \Gamma_1 \rfloor = \lfloor \Gamma \rfloor$, then $\Gamma_1 \sqcap \Gamma, \Gamma_2 \vdash a\{c/z\} :^\ell A$.

Now, LDC($\mathcal{L}$) can be given the exact operational semantics as LDC($\mathcal{Q}_{\mathbb{N}}$). With respect to this operational semantics, LDC($\mathcal{L}$) enjoys the standard type soundness property.

**Theorem 5.12** (Preservation) If $\Gamma \vdash a :^{\ell} A$ and $\vdash a \rightsquigarrow a'$, then $\Gamma \vdash a' :^{\ell} A$.

**Theorem 5.13** (Progress) If $\varnothing \vdash a :^{\ell} A$, then either $a$ is a value or there exists $a'$ such that $\vdash a \rightsquigarrow a'$.

These theorems are not strong enough to show that LDC($\mathcal{L}$) analyses dependencies correctly. For that, we need to show the calculus passes the noninterference test. The noninterference test [Goguen and Meseguer, 1982] ensures that if $\neg(\ell_1 \sqsubseteq \ell_2)$, then variations in $\ell_1$-inputs do not affect $\ell_2$-outputs. In Section 5.3, we shall show that LDC($\mathcal{L}$) passes the noninterference test. But before that, we discuss the relation between Sealing Calculus and LDC($\mathcal{L}$).

### 5.2.5 Sealing Calculus and LDC

The Sealing Calculus [Shikuma and Igarashi, 2006] embeds into LDC($\mathcal{L}$). We don't provide details of the embedding here because LDC($\mathcal{L}$) is essentially the same as SDC. In Chapter 3, we showed that the Sealing Calculus embeds into SDC. Note here that SDC can only be parametrized over lattices and not over semirings that help track linearity.

Given that Sealing Calculus is a general dependency calculus that embeds into LDC($\mathcal{L}$), we conclude that LDC($\mathcal{L}$) is also a general dependency calculus. Thus, LDC can be used for both usage and dependency analyses by parametrizing the calculus over appropriate structures. In the next section, we prove correctness of these analyses in LDC through a heap semantics for the calculus.

## 5.3 Heap Semantics for LDC

LDC models usage of resources and flow of information. Standard operational semantics cannot enforce constraints on usage and flow. However, a heap semantics can do that. In this section, we present a weighted-heap-based semantics for LDC, similar to the one for GRAD in Chapter 4. We use the same heap semantics for analyzing both resource usage and information flow in LDC, just as we used the same standard small-step semantics for both the analyses. The difference between the heap semantics in this chapter and the one in Chapter 4 is that here we use it for both usage and flow analyses whereas there we used it for usage analysis only. Other than this difference, both the semantics are essentially the same.

Recall that heap semantics shows how a term reduces in a heap that assigns values to the free variables of the term. Heaps are ordered lists of variable-term pairs where the terms may be seen as the definitions of the corresponding variables. To every variable-term pair in a heap, we assign a weight, which may be either a $q \in \mathcal{Q}_{\mathbb{N}}$ or an $\ell \in \mathcal{L}$. A heap where every variable-term pair

has a weight associated with it is referred to as a weighted heap. We assume our weighted heaps to be unique, i.e. a variable is not defined twice and acyclic, i.e. definition of a variable does not refer to itself or to other variables appearing subsequently in the heap. We model reduction as an interaction between a term and a weighted heap that defines the free variables of the term.

### 5.3.1 Reduction Relation

The heap-based reduction rules appear in Figure 5.5. There are a few things to note with regard to this reduction relation $[H]a \Longrightarrow_S^q [H']a'$:

- This reduction relation is a simplified version of the one we saw in Chapter 4. We have simplified the relation for a cleaner presentation. Owing to this simplification, we cannot show that certain lemmas hold. But that does not create any problem in proving the soundness theorem. The extra arguments in the reduction relation in Chapter 4 were meant to help one better understand the way reductions happen. Once one gains a good understanding of that, such arguments may be done away with.
- From a resource usage perspective, the judgment above may be read as: $q$ copies of $a$ use resources from heap $H$ to produce $q$ copies of $a'$ with $H'$ being the left-over resources. Regarding a heap as a memory, usage of resources correspond to the number of memory look-ups during reduction.
- From an information flow perspective, the judgment may be read as: under the security constraints of $H$, the term $a$ steps to $a'$ at security level $q$ with $H'$ being the updated security constraints. Regarding a heap as a memory, security labels on assignments correspond to access permissions on data while the label on the judgment corresponds to the security clearance of the user.
- $S$ here denotes a support set of variables that must be avoided while choosing fresh names.

Now, we consider some of the rules presented in Figure 5.5. The most interesting of the rules is rule HeapStep-Var. From a resource usage perspective, this rule may be read as: to step $q$ copies of $x$, we need to look-up the value of $x$ for $q$ times, thereby using up $q$ resources. From an information flow perspective, this rule may be read as: the data pointed to by $x$, residing at security level $r \sqcap q$, is visible to $q$ since $r \sqcap q \sqsubseteq q$. Note that since $r \sqcap q \sqsubseteq r$, the security level of the assignment cannot go down in the updated heap. However, it can always remain the same because $(r \sqcap q) \sqcap q = r \sqcap q$. This rule includes the precondition $q \neq 0$ because at world 0, usage and flow constraints are not meaningful.

The rule HeapStep-Discard is also interesting. From a resource usage perspective, it enables discarding of resources, whenever permitted. From an information flow perspective, it corresponds to information remaining visible to an observer as the observer's security clearance goes up.

Heap, $H ::= \varnothing \mid H, x \overset{q}{\mapsto} a$

$\boxed{[H]a \Longrightarrow_S^q [H']a'}$ *(Selected Heap Step Rules)*

$$\textsc{HeapStep-Var}\ \frac{q \neq 0}{[H_1, x \overset{r+q}{\mapsto} a, H_2]x \Longrightarrow_S^q [H_1, x \overset{r}{\mapsto} a, H_2]a} \qquad \textsc{HeapStep-Discard}\ \frac{[H]a \Longrightarrow_S^q [H']a' \quad q <: q'}{[H]a \Longrightarrow_S^{q'} [H']a'}$$

$$\textsc{HeapStep-AppL}\ \frac{[H]b \Longrightarrow_{S \cup fv\,a}^q [H']b'}{[H]b\ a^r \Longrightarrow_S^q [H']b'\ a^r} \qquad \textsc{HeapStep-AppBeta}\ \frac{y\ \mathit{fresh} \qquad b' = b\{y/x\}}{[H](\lambda^r x : A.b)\ a^r \Longrightarrow_S^q [H, y \overset{q \cdot r}{\mapsto} a]b'}$$

$$\textsc{HeapStep-LetPairBeta}\ \frac{\begin{array}{c} x'\ \mathit{fresh} \qquad y'\ \mathit{fresh} \\ b' = b\{x'/x\}\{y'/y\} \end{array}}{[H]\mathbf{let}_{q_0}\ (x^r, y) = (a_1{}^r, a_2)\ \mathbf{in}\ b \Longrightarrow_S^q [H, x' \overset{q \cdot q_0 \cdot r}{\mapsto} a_1, y' \overset{q \cdot q_0}{\mapsto} a_2]b'}$$

FIGURE 5.5: Heap Semantics for LDC (Excerpt)

With these rules, let us consider some reductions that go through and some that don't.

$$[x \overset{1}{\mapsto} \mathbf{true}]x \Longrightarrow_S^1 [x \overset{0}{\mapsto} \mathbf{true}]\mathbf{true} \text{ but not } [x \overset{0}{\mapsto} \mathbf{true}]x \Longrightarrow_S^1 [x \overset{0}{\mapsto} \mathbf{true}]\mathbf{true}$$
$$[x \overset{2}{\mapsto} \mathbf{true}]x \Longrightarrow_S^2 [x \overset{0}{\mapsto} \mathbf{true}]\mathbf{true} \text{ but not } [x \overset{1}{\mapsto} \mathbf{true}]x \Longrightarrow_S^2 [x \overset{0}{\mapsto} \mathbf{true}]\mathbf{true}$$
$$[x \overset{\mathbf{L}}{\mapsto} \mathbf{true}]x \Longrightarrow_S^{\mathbf{M_1}} [x \overset{\mathbf{L}}{\mapsto} \mathbf{true}]\mathbf{true} \text{ but not } [x \overset{\mathbf{M_2}}{\mapsto} \mathbf{true}]x \Longrightarrow_S^{\mathbf{L}} [x \overset{\mathbf{M_2}}{\mapsto} \mathbf{true}]\mathbf{true}$$

### 5.3.2 Correctness of Usage and Flow

Looking at the rules in Figure 5.5, we observe that they enforce fairness of resource usage. The only rule that allows usage of resources is rule HeapStep-Var. This rule ensures that a look-up goes through only when the environment can provide adequate resources. It also takes away the necessary resources from the environment after a successful look-up. The rules in Figure 5.5 also ensure security of information flow. The only rule that allows information to flow from heap to program is again rule HeapStep-Var. This rule ensures that information can flow through only when the user has the necessary permission.

The following two lemma formalize the arguments presented above. The first lemma says that a definition that is not available at some point during reduction does not become available at a later point. The second lemma says that an unavailable definition does not play any role in reduction. Note that in case of information flow, the constraint $\neg(\exists q_0, r = q + q_0)$ is equivalent to $\neg(r \sqsubseteq q)$. (Here, $|H|$ denotes the length of $H$.) The following lemmas are similar to lemmas 4.6 and 4.7 respectively.

**Lemma 5.14** (Unchanged) If $[H_1, x \overset{r}{\mapsto} a, H_2]c \Longrightarrow^{q}_{S} [H_1', x \overset{r'}{\mapsto} a, H_2']c'$ (where $|H_1| = |H_1'|$) and $\neg(\exists q_0, r = q + q_0)$, then $r' = r$.

**Lemma 5.15** (Irrelevant) If $[H_1, x \overset{r}{\mapsto} a, H_2]c \Longrightarrow^{q}_{S \cup fv\, b} [H_1', x \overset{r'}{\mapsto} a, H_2']c'$ (where $|H_1| = |H_1'|$) and $\neg(\exists q_0, r = q + q_0)$, then $[H_1, x \overset{r}{\mapsto} b, H_2]c \Longrightarrow^{q}_{S \cup fv\, a} [H_1', x \overset{r'}{\mapsto} b, H_2']c'$.

These lemmas, in conjunction with the soundness theorem we present next, show correctness of usage and flow analyses in LDC.

### 5.3.3 Soundness with respect to Heap Semantics

The key idea behind usage analysis through heap semantics is that, if a heap contains the right amount of resources to evaluate some number of copies of a term, as judged by the type system, then the evaluation of that many number of copies of the term in that heap does not get stuck. Since the heap-based reduction rules enforce fairness of resource usage, this would mean that the type system accounts resource usage correctly.

The key idea behind dependency analysis through heap semantics is similar. If a heap sets the right access permissions for a user, as judged by the type system, then the evaluation, in that heap, of any program visible to that user does not get stuck. Since the reduction rules enforce security of information flow, this would mean that the type system allows only secure flows.

The compatibility relation, $H \vDash \Gamma$, between a heap $H$ and a context $\Gamma$, formalizes the idea that the heap $H$ contains the right amount of resources or has set the right access permissions for evaluating any term type-checked in context $\Gamma$. The compatibility relation is similar to the one we saw before:

$\boxed{H \vDash \Gamma}$ *(Compatibility)*

$$\frac{}{\varnothing \vDash \varnothing}\ \textsc{HeapCompat-Empty} \qquad \frac{H \vDash \Gamma_1 + \Gamma_2 \qquad \Gamma_2 \vdash a :^{q} A}{H, x \overset{q}{\mapsto} a \vDash \Gamma_1, x :^{q} A}\ \textsc{HeapCompat-Cons}$$

The soundness theorem stated next says that if a heap $H$ is compatible with a context $\Gamma$, then the evaluation, starting with heap $H$, of a term type-checked in context $\Gamma$ does not get stuck.

**Theorem 5.16** (Soundness) If $H \vDash \Gamma$ and $\Gamma \vdash a :^{q} A$ and $q \neq 0$, then either $a$ is a value or there exists $H', \Gamma', a'$ such that $[H]a \Longrightarrow^{q}_{S} [H']a'$ and $H' \vDash \Gamma'$ and $\Gamma' \vdash a' :^{q} A$.

Below, we present some corollaries of this theorem.

**Lemma 5.17** (No Use) In LDC($\mathcal{Q}_{\mathbb{N}}$): Let $\varnothing \vdash f :^{1} {}^{0}A \to A$. Then, for any $\varnothing \vdash a_1 :^{0} A$ and $\varnothing \vdash a_2 :^{0} A$, the terms $f\ {a_1}^{0}$ and $f\ {a_2}^{0}$ have the same operational behaviour.

The above lemma also holds in LDC($\mathcal{L}$) with 0 and 1 replaced by **H** and **L** respectively. In LDC($\mathcal{L}$), this lemma shows *non-interference* of high-security inputs in low-security outputs.

**Lemma 5.18** (Single Use) In LDC($\mathbb{N}_{\geq}$): Let $\varnothing \vdash f :^1 {}^1A \to A$. Then, for any $\varnothing \vdash a :^1 A$, the term $f\ a^1$ uses $a$ at most once during reduction.

Now that we have seen the syntax and semantics of simply-typed version of LDC, we move on to its Pure Type System (PTS) version.

## 5.4 Linearity and Dependency Analyses in PTS

In this section, we extend LDC to Pure Type Systems. Recall that a Pure Type System is characterized by a tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, where $\mathcal{S}$ is a set of sorts, $\mathcal{A}$ is a set of axioms and $\mathcal{R}$ is a ternary relation between sorts. The PTS version of LDC is similar to its simply-typed version. As far as types and terms are concerned, we just need to add to them the sorts in $\mathcal{S}$ and generalize ${}^rA \to B, {}^rA \times B$ and $A \,\&\, B$ to $\Pi x :^r A.B, \Sigma x :^r A.B$ and $(x : A) \,\&\, B$ respectively. For resource usage and information flow analyses, we use the same parametrizing structures, i.e. $\mathcal{Q}_{\mathbb{N}}$ and $\mathcal{L}$ respectively.

However, there is an important distinction between the simply-typed and PTS versions of LDC. In the PTS version, we need to extend our analyses to dependent types. In Chapters 3 and 4, we have already seen such extensions. In Chapter 3, DDC$^\top$ extends flow analysis in simply-typed SDC to dependent types. In Chapter 4, GRAD extends usage analysis in a graded simply-typed system to dependent types. We follow these extensions in extending the simply-typed version of LDC to dependent types. For usage analysis in dependent types, LDC follows the principle of GRAD. For flow analysis in dependent types, LDC follows the principle of DDC$^\top$. There are, however, some differences between usage analysis in LDC and that in GRAD; we shall point them out below. As far as flow analysis is concerned, LDC and DDC$^\top$ are essentially the same calculus.

We now look at the type system of the PTS version of LDC.

### 5.4.1 Type System of LDC

The typing and equality rules appear in Figures 5.6 and 5.7 respectively. There are a few things to note:

- We use these rules for both resource usage and information flow analyses.
- The judgment $\Delta \vdash_0 a : A$ is shorthand for the judgment $\Gamma \vdash a :^0 A$ where $\lfloor \Gamma \rfloor = \Delta$ and $\overline{\Gamma}$ is a 0 vector. Note that this judgment is essentially the standard typing judgment $\Delta \vdash a : A$ because in world 0, neither resource usage nor information flow constraints are meaningful.

$\boxed{\Gamma \vdash a :^q A}$ *(PTS version)*

PTS-Axiom
$$\frac{c : s \in \mathcal{A}}{\varnothing \vdash c :^q s}$$

PTS-Var
$$\frac{\Delta \vdash_0 A : s \qquad \lfloor \Gamma \rfloor = \Delta}{0 \cdot \Gamma, x :^q A \vdash x :^q A}$$

PTS-Weak
$$\frac{\Gamma \vdash a :^q A \qquad \Delta \vdash_0 B : s \qquad \lfloor \Gamma \rfloor = \Delta}{\Gamma, y :^0 B \vdash a :^q A}$$

PTS-Pi
$$\frac{\Gamma_1 \vdash A :^q s_1 \qquad \Gamma_2, x :^{r_0} A \vdash B :^q s_2 \qquad \mathcal{R}(s_1, s_2, s_3) \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash \Pi x :^r A.B :^q s_3}$$

PTS-Lam
$$\frac{\Gamma, x :^{q \cdot r} A \vdash b :^q B \qquad \Delta \vdash_0 \Pi x :^r A.B : s \qquad \lfloor \Gamma \rfloor = \Delta}{\Gamma \vdash \lambda^r x : A.b :^q \Pi x :^r A.B}$$

PTS-App
$$\frac{\Gamma_1 \vdash b :^q \Pi x :^r A.B \qquad \Gamma_2 \vdash a :^{q \cdot r} A \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash b\ a^r :^q B\{a/x\}}$$

PTS-Conv
$$\frac{\Gamma \vdash a :^q A \qquad \Delta \vdash_0 B : s \qquad A =_\beta B \qquad \lfloor \Gamma \rfloor = \Delta}{\Gamma \vdash a :^q B}$$

PTS-SubL
$$\frac{\Gamma' \vdash a :^q A \qquad \Gamma <: \Gamma'}{\Gamma \vdash a :^q A}$$

PTS-SubR
$$\frac{\Gamma \vdash a :^q A \qquad q <: q'}{\Gamma \vdash a :^{q'} A}$$

PTS-WPair
$$\frac{\Delta \vdash_0 \Sigma x :^r A_1.A_2 : s \qquad \Gamma_1 \vdash a_1 :^{q \cdot r} A_1 \qquad \Gamma_2 \vdash a_2 :^q A_2\{a_1/x\} \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor = \Delta}{\Gamma_1 + \Gamma_2 \vdash (a_1{}^r, a_2) :^q \Sigma x :^r A_1.A_2}$$

PTS-LetPair
$$\frac{\Delta, z : \Sigma x :^r A_1.A_2 \vdash_0 B : s \qquad \Gamma_1 \vdash a :^{q \cdot q_0} \Sigma x :^r A_1.A_2 \qquad \Gamma_2, x :^{q \cdot q_0 \cdot r} A_1, y :^{q \cdot q_0} A_2 \vdash b :^q B\{(x^r, y)/z\} \qquad q_0 <: 1 \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor = \Delta}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}_{q_0}\ (x^r, y) = a\ \mathbf{in}\ b :^q B\{a/z\}}$$

PTS-SPair
$$\frac{\Delta \vdash_0 (x : A_1)\ \&\ A_2 : s \qquad \Gamma \vdash a_1 :^q A_1 \qquad \Gamma \vdash a_2 :^q A_2\{a_1/x\} \qquad \lfloor \Gamma \rfloor = \Delta}{\Gamma \vdash (a_1; a_2) :^q (x : A_1)\ \&\ A_2}$$

PTS-Proj1
$$\frac{\Delta \vdash_0 (x : A_1)\ \&\ A_2 : s \qquad \Gamma \vdash a :^q (x : A_1)\ \&\ A_2 \qquad \lfloor \Gamma \rfloor = \Delta}{\Gamma \vdash \mathbf{proj}_1\ a :^q A_1}$$

PTS-Proj2
$$\frac{\Delta \vdash_0 (x : A_1)\ \&\ A_2 : s \qquad \Gamma \vdash a :^q (x : A_1)\ \&\ A_2 \qquad \lfloor \Gamma \rfloor = \Delta}{\Gamma \vdash \mathbf{proj}_2\ a :^q A_2\{\mathbf{proj}_1\ a/x\}}$$

PTS-Sum
$$\frac{\Gamma_1 \vdash A_1 :^q s \qquad \Gamma_2 \vdash A_2 :^q s \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor}{\Gamma_1 + \Gamma_2 \vdash A_1 + A_2 :^q s}$$

PTS-Case
$$\frac{\Delta, z : A_1 + A_2 \vdash_0 B : s \qquad \Gamma_1 \vdash a :^{q \cdot q_0} A_1 + A_2 \qquad \Gamma_2, x_1 :^{q \cdot q_0} A_1 \vdash b_1 :^q B\{\mathbf{inj}_1\ x_1/z\} \qquad \Gamma_2, x_2 :^{q \cdot q_0} A_2 \vdash b_2 :^q B\{\mathbf{inj}_2\ x_2/z\} \qquad q_0 <: 1 \qquad \lfloor \Gamma_1 \rfloor = \lfloor \Gamma_2 \rfloor = \Delta}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_{q_0}\ a\ \mathbf{of}\ x_1.b_1\ ;\ x_2.b_2 :^q B\{a/z\}}$$

Figure 5.6: Type System for LDC (Excerpt)

$\boxed{A =_\beta B}$ *(Definitional Equality)*

Eq-PiCong
$$\frac{A_1 =_\beta A_2 \qquad B_1 =_\beta B_2}{\Pi x :^r A_1.B_1 =_\beta \Pi x :^r A_2.B_2}$$

Eq-LamCong
$$\frac{A_1 =_\beta A_2 \qquad b_1 =_\beta b_2}{\lambda^r x : A_1.b_1 =_\beta \lambda^r x : A_2.b_2}$$

Eq-AppCong
$$\frac{b_1 =_\beta b_2 \qquad a_1 =_\beta a_2}{b_1\ a_1{}^r =_\beta b_2\ a_2{}^r}$$

Figure 5.7: Equality Rules for LDC (Excerpt)

- The type system uses the judgment $\Delta \vdash_0 A : s$ to check well-formedness of types in several rules, for example, rule PTS-Lam, rule PTS-WPair, etc. In this regard, LDC is different from GraD which does not have the judgment form $\Gamma \vdash a :^0 A$. Recall that the grade to the right of the turnstile in every typing judgment in GraD is implicitly 1.
- The type system tracks usage (and flow) in terms and types separately. The rule PTS-Var illustrates this principle nicely. The type $A$ may use some resources or be visible at some low security level. But while type-checking a term of type $A$, we zero-out the requirements of $A$ or set $A$ to the highest security level. This principle also applies to several other rules, for example, rule PTS-Lam, rule PTS-WPair, etc.
- Like QTT [Atkey, 2018] and unlike GraD, LDC supports strong $\Sigma$-types.
- Like GraD and DDC$^\top$, LDC uses $\beta$-equivalence for equality in rule PTS-Conv. It is a congruent, equivalence relation closed under $\beta$-reduction of terms. Some of the equality rules appear in Figure 5.7. They are mostly standard. However, the congruence rules Eq-PiCong, Eq-LamCong, and Eq-AppCong need to check that the grade annotations on the terms being equated match up.

Next, we look at the metatheory of the calculus.

### 5.4.2 Metatheory of LDC

The PTS version of LDC satisfies the PTS analogues of all the lemmas and theorems satisfied by the simply-typed version, presented in Sections 5.2.2 and 5.2.4. The PTS version also enjoys the same standard operational semantics as the simply-typed version. Further, the PTS version is type-sound with respect to this semantics.

Next, we state the PTS analogues of some of the crucial lemmas and theorems presented in Sections 5.2.2 and 5.2.4.

**Lemma 5.19** (Weakening) If $\Gamma_1, \Gamma_2 \vdash a :^q A$ and $\Delta_1 \vdash_0 C : s$ and $\lfloor \Gamma_1 \rfloor = \Delta_1$, then $\Gamma_1, z :^0 C, \Gamma_2 \vdash a :^q A$.

**Lemma 5.20** (Substitution) If $\Gamma_1, z :^{r_0} C, \Gamma_2 \vdash a :^q A$ and $\Gamma \vdash c :^{r_0} C$ and $\lfloor \Gamma_1 \rfloor = \lfloor \Gamma \rfloor$, then $\Gamma_1 + \Gamma, \Gamma_2\{c/z\} \vdash a\{c/z\} :^q A\{c/z\}$.

**Theorem 5.21** (Preservation) If $\Gamma \vdash a :^q A$ and $\vdash a \rightsquigarrow a'$, then $\Gamma \vdash a' :^q A$.

**Theorem 5.22** (Progress) If $\varnothing \vdash a :^q A$, then either $a$ is a value or there exists $a'$ such that $\vdash a \rightsquigarrow a'$.

Now, we consider heap semantics for LDC.

### 5.4.3 Heap Semantics for LDC

The PTS version of LDC enjoys the same heap reduction relation as the simply-typed version. However, the presence of dependent types causes the same issue with heap semantics that we saw in Section 4.6. Because substitutions are delayed through the heap, the terms and their types 'get out of sync'. To overcome this issue, we follow the same strategy we used in Section 4.6. We extend the type system of LDC with contexts that allow definitions. Then, we show that the extended calculus is sound with respect to heap semantics. Thereafter, we prove that the original calculus can be elaborated into the extended calculus. Via this elaboration, we conclude that the original calculus is sound with respect to heap semantics. We omit the technical details of this construction because they are almost the same as in Section 4.6.

LDC is sound with respect to heap semantics. Note the statement below is the same as its simply-typed counterpart.

**Theorem 5.23** (Soundness) If $H \vDash \Gamma$ and $\Gamma \vdash a :^q A$ and $q \neq 0$, then either $a$ is a value or there exists $H', \Gamma', a'$ such that $[H]a \Longrightarrow_S^q [H']a'$ and $H' \vDash \Gamma'$ and $\Gamma' \vdash a' :^q A$.

Below, we present some corollaries of this theorem.

**Corollary 5.24** In LDC($\mathcal{Q}_{\mathbb{N}}$): If $\varnothing \vdash f :^1 \Pi x :^0 s.x$ and $\varnothing \vdash_0 A : s$, then $f\ A^0$ must diverge.

**Corollary 5.25** In LDC($\mathcal{Q}_{\mathbb{N}}$): In a strongly normalizing PTS, if $\varnothing \vdash f :^1 \Pi x :^0 s.\Pi y :^1 x.x$ and $\varnothing \vdash_0 A : s$ and $\varnothing \vdash a :^1 A$, then $f\ A^0\ a^1 =_\beta a$.

The proofs of these corollaries appear in Choudhury [2022c].

## 5.5 Adding Unrestricted Use

Till now, we used LDC for analyzing exact use, bounded use and dependency. In this section, we use the calculus for analyzing unrestricted use as well. Unrestricted use, referred to by $\omega$, is different from exact and bounded use, referred to by $n \in \mathbb{N}$, in two ways:

- $\omega$ is an additive annihilator, meaning $\omega + q = q + \omega = \omega$, for $q \in \mathbb{N} \cup \{\omega\}$.
  No $n \in \mathbb{N}$ is an additive annihilator.
- $\omega$ is a multiplicative annihilator (almost) as well, meaning $\omega \cdot q = q \cdot \omega = \omega$ for $q \in (\mathbb{N} - \{0\}) \cup \{\omega\}$. No $n \in \mathbb{N} - \{0\}$ is a multiplicative annihilator.

To accommodate this behaviour of $\omega$, we need to make a change to our type system. But before we make this change, let us fix our preordered semirings:

- $\mathbb{N}^\omega_=$, that contains the preordered semiring $\mathbb{N}_=$ and $\omega$ with $\omega <: q$ for all $q$
- $\mathbb{N}^\omega_\geq$, that contains the preordered semiring $\mathbb{N}_\geq$ and $\omega$ with $\omega <: q$ for all $q$

- $\mathcal{Q}_{\text{Lin}}$ and $\mathcal{Q}_{\text{Aff}}$, described in Section 5.1.4

We use $\mathcal{Q}^{\omega}_{\mathbb{N}}$ to denote an arbitrary member of the above set of semirings. Next, we discuss the change necessary as we move from $\text{LDC}(\mathcal{Q}_{\mathbb{N}})$ to $\text{LDC}(\mathcal{Q}^{\omega}_{\mathbb{N}})$.

### 5.5.1 A Problem and its Solution

When unrestricted use is allowed, the type systems in Figures 5.2 and 5.6 cannot enforce fairness of resource usage. Consider the following 'unfair' derivation allowed by the simple type system:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x :^{\omega} A \vdash x :^{\omega} A}\ \text{ST-Var} \quad \dfrac{}{x :^{\omega} A \vdash x :^{\omega} A}\ \text{ST-Var}}{x :^{\omega} A \vdash (x^1, x) :^{\omega} {}^1A \times A}\ \text{ST-WPair}}{\varnothing \vdash \lambda^1 x : A.(x^1, x) :^{\omega} {}^1A \to {}^1A \times A}\ \text{ST-Lam}}{\varnothing \vdash \lambda^1 x : A.(x^1, x) :^{1} {}^1A \to {}^1A \times A}\ \text{ST-SubR}$$

The judgment $\varnothing \vdash \lambda^1 x : A.(x^1, x) :^1 {}^1A \to {}^1A \times A$ is unfair because it allows copying of resources. Carefully observing the derivation, we find that the unfairness arises when $\omega$ 'tricks' the ST-Lam rule into believing that the term uses $x$ (at most) once.

This unfairness leads to a failure in type soundness. To see how, consider the term: $y :^1 A \vdash (\lambda^1 x : A.(x^1, x))\ y^1 :^1 {}^1A \times A$ that type-checks via the above derivation and rule ST-App. This term steps to: $\vdash (\lambda^1 x : A.(x^1, x))\ y^1 \rightsquigarrow (y^1, y)$. But then, we have unsoundness because: $y :^1 A \nvdash (y^1, y) :^1 {}^1A \times A$. Therefore, to ensure type soundness, we need to modify rule ST-Lam and rule PTS-Lam.

We modify these rules as follows:

$$\text{ST-LamOmega}\ \dfrac{\Gamma, x :^{q \cdot r} A \vdash b :^{q} B \qquad q = \omega \Rightarrow r = \omega \qquad q_0 \neq 0}{q_0 \cdot \Gamma \vdash \lambda^r x : A.b :^{q_0 \cdot q} {}^rA \to B}$$

$$\text{PTS-LamOmega}\ \dfrac{\Gamma, x :^{q \cdot r} A \vdash b :^{q} B \qquad \Delta \vdash_0 \Pi x :^{r} A.B : s \qquad \lfloor \Gamma \rfloor = \Delta \qquad q = \omega \Rightarrow r = \omega \qquad q_0 \neq 0}{q_0 \cdot \Gamma \vdash \lambda^r x : A.b :^{q_0 \cdot q} \Pi x :^{r} A.B}$$

There are several points to note here:

- Rules ST-LamOmega and PTS-LamOmega are generalizations of rules ST-Lam and PTS-Lam respectively.
- These rules impose the constraint: $q = \omega \Rightarrow r = \omega$. This way $\omega$ won't be able to 'trick' the Lambda-rule into believing that functions use their arguments less than what they actually do. In particular, with these modified rules, the above unfair derivation won't go through.

- The constraint $q = \omega \Rightarrow r = \omega$, while required for blocking unfair derivations, also blocks some fair ones like the one below:

$$\frac{x :^{\omega} A \vdash x :^{\omega} A}{\varnothing \vdash \lambda^{1} x : A.x :^{\omega} {}^{1}A \to A}$$

  To allow such derivations, while still imposing this constraint, rules ST-LamOmega and PTS-LamOmega multiply the conclusion judgment by $q_0$. This multiplication helps these rules allow the above derivation as:

$$\frac{x :^{1} A \vdash x :^{1} A}{\varnothing \vdash \lambda^{1} x : A.x :^{\omega} {}^{1}A \to A}$$

- The side condition $q_0 \neq 0$ makes sure that a meaningful judgment is not turned into a meaningless one. Recall that judgments in 0-world are meaningless, as far as usage and dependency analyses are concerned.

- The rules ST-LamOmega and PTS-LamOmega may seem specific to usage analysis because the constraint $q = \omega \Rightarrow r = \omega$ does not have an analogue in dependency analysis. This, however, is a minor issue because we can rephrase this constraint as: $(q + q = q) \wedge (q \cdot r = q) \Rightarrow r <: q$.
  Wrt usage analysis, the above two constraints are equivalent.
  Wrt dependency analysis, the latter constraint is a tautology because $q \sqcup r = q \Rightarrow r \sqsubseteq q$. So rules ST-LamOmega and PTS-LamOmega make sense from the perspective of dependency analysis as well. Further, when viewed as rules analysing dependency, rules ST-LamOmega and PTS-LamOmega are equivalent to rules ST-Lam and PTS-Lam respectively.

- Rule ST-LamOmega can also be equivalently replaced with the following two simpler rules (and similarly for rule PTS-LamOmega):

$$\text{ST-LamOmega0}\ \frac{\Gamma, x :^{q \cdot r} A \vdash b :^{q} B \qquad q = \omega \Rightarrow r = \omega}{\Gamma \vdash \lambda^{r} x : A.b :^{q\ r} A \to B} \qquad \text{ST-Omega}\ \frac{\omega \cdot \Gamma \vdash a :^{q} A \qquad q \neq 0}{\omega \cdot \Gamma \vdash a :^{\omega} A}$$

The above modification is the only change that we need to make to the type systems presented in Figures 5.2 and 5.6 in order to track unrestricted use.

With this modification, LDC($\mathcal{Q}^{\omega}_{\mathbb{N}}$) satisfies all the lemmas and theorems satisfied by LDC($\mathcal{Q}_{\mathbb{N}}$). In particular, LDC($\mathcal{Q}^{\omega}_{\mathbb{N}}$) satisfies type soundness (Theorems 5.21 and 5.22) and heap soundness (Theorem 5.23). We state this property as a theorem below.

**Theorem 5.26** LDC($\mathcal{Q}^{\omega}_{\mathbb{N}}$) satisfies type soundness and heap soundness.

$$\overline{1} = \mathbf{Unit} \qquad \overline{X \times Y} = \overline{X} \,\&\, \overline{Y} \qquad \overline{I} = \mathbf{Unit} \qquad \overline{A \otimes B} = {}^{1}\overline{A} \times \overline{B}$$
$$\overline{X \to Y} = {}^{\omega}\overline{X} \to \overline{Y} \qquad \overline{G\,A} = \overline{A} \qquad \overline{A \multimap B} = {}^{1}\overline{A} \to \overline{B} \qquad \overline{F\,X} = {}^{\omega}\overline{X} \times \mathbf{Unit}$$

$$\overline{x} = x \qquad \overline{a} = a \qquad \overline{()} = \mathbf{unit} \qquad \overline{*} = \mathbf{unit}$$
$$\overline{(s,t)} = (\overline{s};\overline{t}) \qquad \overline{e \otimes f} = (\overline{e}^{1}, \overline{f})$$
$$\overline{\mathbf{fst}(s)} = \mathbf{proj}_1\,\overline{s} \qquad \overline{\mathbf{let}\ a \otimes b = e\ \mathbf{in}\ f} = \mathbf{let}_1\ (a^{1}, b) = \overline{e}\ \mathbf{in}\ \overline{f}$$
$$\overline{\mathbf{snd}(s)} = \mathbf{proj}_2\,\overline{s} \qquad \overline{\mathbf{let} * = e\ \mathbf{in}\ f} = \mathbf{let}_1\ \mathbf{unit} = \overline{e}\ \mathbf{in}\ \overline{f}$$
$$\overline{\lambda x : X.s} = \lambda^{\omega} x : \overline{X}.\overline{s} \qquad \overline{\lambda a : A.e} = \lambda^{1} a : \overline{A}.\overline{e} \qquad \overline{s\,t} = \overline{s}\,\overline{t}^{\omega} \qquad \overline{e\,f} = \overline{e}\,\overline{f}^{1}$$
$$\overline{G\,e} = \overline{e} \qquad \overline{\mathbf{derelict}\ s} = \overline{s}$$
$$\overline{F\,s} = (\overline{s}^{\omega}, \mathbf{unit}) \qquad \overline{\mathbf{let}\ Fx = e\ \mathbf{in}\ f} = \mathbf{let}_1\ (x^{\omega}, y) = \overline{e}\ \mathbf{in}\ \overline{f} \quad (y \text{ fresh})$$

FIGURE 5.8: Type and term translation from LNL $\lambda$-calculus to LDC($\mathcal{Q}_{\text{Lin}}$)

Thus, LDC is a general linearity and dependency calculus. For tracking linearity in LDC, we can use any of the $\mathcal{Q}^{\omega}_{\mathbb{N}}$s. For tracking dependency in LDC, we can use any lattice. For tracking linearity and dependency simultaneously in LDC, we can use the cartesian product of these structures.

## 5.6 LDC vs. Standard Linear and Dependency Calculi

In section 5.2.5, we discussed how Sealing Calculus, a standard dependency calculus, embeds into LDC($\mathcal{L}$). Now, we compare LDC with a standard linear calculus, the Linear Nonlinear (LNL) $\lambda$-calculus of Benton [1994]. The LNL $\lambda$-calculus tracks just linear and unrestricted use and is simply-typed. So we compare it with simply-typed LDC($\mathcal{Q}_{\text{Lin}}$).

The LNL calculus employs two types of contexts and two types of typing judgments. The two types of contexts, linear and nonlinear, correspond to assumptions at grades 1 and $\omega$ respectively in LDC($\mathcal{Q}_{\text{Lin}}$). The two types of judgments, linear and nonlinear, correspond to derivations in worlds 1 and $\omega$ respectively in LDC($\mathcal{Q}_{\text{Lin}}$). Note that in LNL calculus, linear and nonlinear contexts are denoted by $\Gamma$ and $\Theta$ respectively; linear and nonlinear judgments are written as $\Theta;\Gamma \vdash_{\mathcal{L}} e : A$ and $\Theta \vdash_{\mathcal{C}} t : X$ respectively. The calculus contains standard intuitionistic types and linear types; it also contains two type constructors, $F$ and $G$, via which the linear and the nonlinear worlds interact. The calculus uses $A, B$ for linear types; $X$, $Y$ for nonlinear types; $a, b$ for linear variables; $x, y$ for nonlinear variables; $e, f$ for linear terms; $s, t$ for nonlinear terms.

We present the translation function from LNL $\lambda$-calculus [Benton, 1994] to LDC($\mathcal{Q}_{\text{Lin}}$) in Figure 5.8. This translation preserves typing and meaning.

**Theorem 5.27** If $\Theta; \Gamma \vdash_{\mathcal{L}} e : A$, then $\overline{\Theta}^{\omega}, \overline{\Gamma}^{1} \vdash \overline{e} :^{1} \overline{A}$.
If $\Theta \vdash_{\mathcal{C}} t : X$, then $\overline{\Theta}^{\omega} \vdash \overline{t} :^{\omega} \overline{X}$.
If $e =_{\beta} f$, then $\overline{e} =_{\beta} \overline{f}$. If $s =_{\beta} t$ then $\overline{s} =_{\beta} \overline{t}$.

Here, $\overline{\Gamma}^{1}$ and $\overline{\Theta}^{\omega}$ denote $\Gamma$ and $\Theta$, with the types translated, and assumptions held at 1 and $\omega$ respectively. Further, $\_ =_{\beta} \_$ denotes the beta equivalence relation on the terms of LNL calculus [Benton and Wadler, 1996].

Note that a translation in the other direction from LDC($\mathcal{Q}_{\mathrm{Lin}}$) to LNL calculus would fail because the latter does not model 0-use. Next, we formally compare LDC with GRAD and DDC$^{\top}$. GRAD is parametrized by an arbitrary preordered semiring. LDC($\mathcal{Q}^{\omega}_{\mathbb{N}}$), on the other hand, is parametrized by specific preordered semirings, i.e. $\mathcal{Q}^{\omega}_{\mathbb{N}}$s. When compared over these semirings, we can show that LDC subsumes GRAD. We can also show that over arbitrary lattices, LDC subsumes DDC$^{\top}$.

**Theorem 5.28** With $\mathcal{Q}^{\omega}_{\mathbb{N}}$ as the parametrizing structure, if $\Gamma \vdash a : A$ in GRAD, then $\Gamma \vdash a :^{1} A$ in LDC. Further, if $\vdash a \rightsquigarrow a'$ in GRAD, then $\vdash a \rightsquigarrow a'$ in LDC.

**Theorem 5.29** With $\mathcal{L}$ as the parametrizing structure, if $\Gamma \vdash a :^{\ell} A$ in DDC$^{\top}$, then $\Gamma \vdash a :^{\ell} A$ in LDC. Further, if $\vdash a \rightsquigarrow a'$ in DDC$^{\top}$, then $\vdash a \rightsquigarrow a'$ in LDC.

## 5.7 Conclusion

In this chapter, we have shown that linearity and dependency analyses can be systematically unified and combined into a single calculus. We presented, LDC, a general linearity and dependency calculus parametrized by an arbitrary PTS. We showed that linearity and dependency analyses in LDC are correct using a heap semantics. We also showed that LDC subsumes standard calculi for linearity and dependency analyses. Here, we focused on the syntactic properties of LDC. In a future work, we plan to explore the semantic properties of the calculus. In particular, we want to find out how semantic models of LDC compare with the categorical models of linear and dependency type systems.

# Chapter 6

# Conclusion

In this dissertation, we have designed calculi for dependency and linearity analyses in Pure Type Systems. We first designed $\text{GMCC}_\text{e}$, a calculus for dependency analysis in simple type systems. $\text{GMCC}_\text{e}$ has a nice categorical interpretation and subsumes standard dependency calculi from literature. We extended $\text{GMCC}_\text{e}$ to $\text{DDC}^\top$ and DDC. Both $\text{DDC}^\top$ and DDC can analyze dependencies in Pure Type Systems. However, DDC is more general than $\text{DDC}^\top$ because not only does it analyze dependencies but also makes use of dependency analysis in its type system itself. We use $\text{DDC}^\top$ and DDC to analyze fine-grained notions of irrelevance in dependently-typed programs. Next, we designed GRAD, a calculus for linearity and other quantitative analyses in Pure Type Systems. We use GRAD to reason about various forms of resource usage in dependently-typed programs, for example, no use, linear use, etc. Thereafter, we designed LDC, a calculus for combined linearity and dependency analysis in Pure Type Systems. LDC, essentially an integration of GRAD and $\text{DDC}^\top$, can reason about both usage and dependencies. This dissertation establishes the thesis that these calculi provide a systematic way for carrying out dependency, linearity or their combined analysis in any Pure Type System.

Though the focus of this dissertation has solely been dependency and linearity analyses, we believe that the ideas introduced here would be useful for other analyses in Pure Type Systems as well. The reason behind this belief is that in essence, all our calculi are modeling interactions between programs and their environments via grades. As such, these calculi may be adapted to analyze interactions other than dependency and linearity.

# Bibliography


Martín Abadi. 2006. Access Control in a Core Calculus of Dependency. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. Association for Computing Machinery, New York, NY, USA, 263–273. https://doi.org/10.1145/1159803.1159839

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 147–160. https://doi.org/10.1145/292540.292555

Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. 1996. Analysis and Caching of Dependencies. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming* (Philadelphia, Pennsylvania, USA) *(ICFP '96)*. Association for Computing Machinery, New York, NY, USA, 83–91. https://doi.org/10.1145/232627.232638

Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. *Proc. ACM Program. Lang.* 4, ICFP, Article 90 (Aug. 2020), 28 pages. https://doi.org/10.1145/3408972

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (mar 2012). https://doi.org/10.2168/lmcs-8(1:29)2012

Samson Abramsky. 1993. Computational interpretations of linear logic. *Theoretical Computer Science* 111, 1 (1993), 3–57. https://doi.org/10.1016/0304-3975(93)90181-R

The Agda Team. 2021. *Agda Standard Library.* Retrieved October 23, 2022 from https://agda.github.io/agda-stdlib/Data.Fin.Base.html

Maximilian Algehed. 2018. A Perspective on the Dependency Core Calculus. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security* (Toronto, Canada) *(PLAS '18)*. Association for Computing Machinery, New York, NY, USA, 24–28. https://doi.org/10.1145/3264820.3264823

Maximilian Algehed and Jean-Philippe Bernardy. 2019. Simple Noninterference from Parametricity. *Proc. ACM Program. Lang.* 3, ICFP, Article 89 (July 2019), 22 pages. https://doi.org/10.1145/3341693

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 56–65. https://doi.org/10.1145/3209108.3209189

Andrew Barber. 1996. *Dual Intuitionistic Linear Logic.* Technical Report. Edinburgh, Scotland.

H. P. Barendregt. 1993. *Lambda Calculi with Types.* Oxford University Press, Inc., USA, 117–309.

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *Foundations of Software Science and Computational Structures*, Roberto Amadio (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 365–379.

N. Benton and P. Wadler. 1996. Linear logic, monads and the lambda calculus. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. 420–431. https://doi.org/10.1109/LICS.1996.561458

P. N. Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract). In *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL '94)*. Springer-Verlag, Berlin, Heidelberg, 121–135.

P. N. Benton, Gavin M. Bierman, Valeria de Paiva, and Martin Hyland. 1993. A Term Calculus for Intuitionistic Linear Logic. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA '93)*. Springer-Verlag, Berlin, Heidelberg, 75–90.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. In *Principles of Programming Languages 2018 (POPL 2018)*.

Jean-Philippe Bernardy and Moulin Guilhem. 2013. Type-Theory in Color. *SIGPLAN Not.* 48, 9 (Sept. 2013), 61–72. https://doi.org/10.1145/2544174.2500577

G. Birkhoff. 1967. *Lattice Theory* (3rd ed.). American Mathematical Society, Providence.

William J. Bowman and Amal Ahmed. 2015. Noninterference for Free. *SIGPLAN Not.* 50, 9 (Aug. 2015), 101–113. https://doi.org/10.1145/2858949.2784733

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Stephen Brookes and Shai Geva. 1992. Computational Comonads and Intensional Semantics. In *Applications of Categories in Computer Science (Proceedings of the LMS Symposium, Vol. 177)*. Cambridge University Press.

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear logic propositions as session types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423. https://doi.org/10.1017/S0960129514000218

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering*, Frank Pfenning and Yannis Smaragdakis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 57–76.

Jawahar Chirimar, Carl Gunter, and Jon Riecke. 2000. Reference Counting as a Computational Interpretation of Linear Logic. *Journal of Functional Programming* 6 (03 2000). https://doi.org/10.1017/S0956796800001660

Pritam Choudhury. 2022a. Monadic and Comonadic Aspects of Dependency Analysis. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 172 (Oct. 2022), 29 pages. https://doi.org/10.1145/3563335

Pritam Choudhury. 2022b. Monadic and Comonadic Aspects of Dependency Analysis. https://doi.org/10.48550/ARXIV.2209.06334

Pritam Choudhury. 2022c. Unifying Linearity and Dependency Analyses. https://github.com/pritamChoudhury/UnderReviewPaper Under Review.

Pritam Choudhury, Harley Eades, Richard A. Eisenberg, and Stephanie C Weirich. 2020. A graded dependent type system with a usage-aware semantics (extended version). https://doi.org/10.48550/ARXIV.2011.04070

Pritam Choudhury, Harley Eades, and Stephanie Weirich. 2022a. A Dependent Dependency Calculus. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 403–430.

Pritam Choudhury, Harley Eades, and Stephanie Weirich. 2022b. A Dependent Dependency Calculus (Extended Version). https://doi.org/10.48550/ARXIV.2201.11040

Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A Graded Dependent Type System with a Usage-Aware Semantics. *Proc. ACM Program. Lang.* 5, POPL, Article 50 (Jan. 2021), 32 pages. https://doi.org/10.1145/3434331

Rowan Davies. 2017. A Temporal Logic Approach to Binding-Time Analysis. *J. ACM* 64, 1, Article 1 (mar 2017), 45 pages. https://doi.org/10.1145/3011069

Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243. https://doi.org/10.1145/360051.360056

Samuel Eilenberg and G. Max Kelly. 1966. Closed Categories. In *Proceedings of the Conference on Categorical Algebra*, S. Eilenberg, D. K. Harrison, S. MacLane, and H. Röhrl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 421–562.

Richard A. Eisenberg, Guillaume Duboc, Stephanie Weirich, and Daniel Lee. 2021. An Existential Crisis Resolved: Type Inference for First-Class Existential Types. *Proc. ACM Program. Lang.* 5, ICFP, Article 64 (aug 2021), 29 pages. https://doi.org/10.1145/3473569

Soichiro Fujii. 2019. A 2-Categorical Study of Graded and Indexed Monads. arXiv:1904.08083 [math.CT]

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining Effects and Coeffects via Grading. *SIGPLAN Not.* 51, 9 (sep 2016), 476–489. https://doi.org/10.1145/3022670.2951939

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 331–350.

David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 28–38. https://doi.org/10.1145/319838.319848

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

Jean-Yves Girard. 1995. Linear Logic: Its Syntax and Semantics. In *Proceedings of the Workshop on Advances in Linear Logic*. Cambridge University Press, USA, 1–42.

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science* 97, 1 (1992), 1–66. https://doi.org/10.1016/0304-3975(92)90386-T

J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11.

Jonathan S. Golan. 1999. *Semirings and their Applications*. Springer Netherlands. https://doi.org/10.1007/978-94-015-9333-5

Carsten K. Gomard and Neil D. Jones. 1991. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming* 1, 1 (1991), 21–69. https://doi.org/10.1017/S0956796800000058

John Hatcliff and Olivier Danvy. 1997. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science* 7, 5 (1997), 507–541. https://doi.org/10.1017/S0960129597002405

Nevin Heintze and Jon G. Riecke. 1998. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 365–377. https://doi.org/10.1145/268946.268976

Bart Jacobs. 1999. *Categorical Logic and Type Theory*. Elsevier, Amsterdam, The Netherlands.

Anita K. Jones and Richard J. Lipton. 1975. The Enforcement of Security Policies for Computation. *SIGOPS Oper. Syst. Rev.* 9, 5 (nov 1975), 197–206. https://doi.org/10.1145/1067629.806538

Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. *SIGPLAN Not.* 49, 1 (jan 2014), 633–645. https://doi.org/10.1145/2578855.2535846

G. A. Kavvos. 2019. Modalities, Cohesion, and Information Flow. *Proc. ACM Program. Lang.* 3, POPL, Article 20 (jan 2019), 29 pages. https://doi.org/10.1145/3290333

Anders Kock. 1970. Monads on symmetric monoidal closed categories. 21 (1970), 1–10. https://doi.org/10.1007/BF01220868

Anders Kock. 1972. Strong Functors and Monoidal Monads. 23 (1972), 113–120. https://doi.org/10.1007/BF01304852

Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/2676726.2676969

Y. Lafont. 1988. The linear abstract machine. *Theoretical Computer Science* 59, 1 (1988), 157–180. https://doi.org/10.1016/0304-3975(88)90100-4

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 144–154. https://doi.org/10.1145/158511.158618

P. Lincoln and J. Mitchell. 1992. Operational aspects of linear lambda calculus. In *1992 Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, Los Alamitos, CA, USA, 235,236,237,238,239,240,241,242,243,244,245,246. https://doi.org/10.1109/LICS.1992.185536

Luísa Lourenço and Luís Caires. 2015. Dependent Information Flow Types. *SIGPLAN Not.* 50, 1 (Jan. 2015), 317–328. https://doi.org/10.1145/2775051.2676994

Saunders MacLane. 1971. *Categories for the Working Mathematician.* Springer-Verlag, New York. ix+262 pages. Graduate Texts in Mathematics, Vol. 5.

Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science VI*, L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 104. Elsevier, 153–175. https://doi.org/10.1016/S0049-237X(09)70189-2

Conor McBride. 2016. *I Got Plenty o' Nuttin'.* Springer International Publishing, Cham, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Alexandre Miquel. 2001. The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In *Typed Lambda Calculi and Applications*, Samson Abramsky (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 344–359.

Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Foundations of Software Science and Computational Structures* (Budapest, Hungary) *(FOSSACS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 350–364.

Richard Nathan Mishra-Linger. 2008. *Irrelevance, Polymorphism, and Erasure in Type Theory.* Ph.D. Dissertation. Portland State University, Department of Computer Science. https://doi.org/10.15760/etd.2669

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. https://www.sciencedirect.com/science/article/pii/0890540191900524 Selections from 1989 IEEE Symposium on Logic in Computer Science.

Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. In *Programming Languages and Systems*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 462–490.

Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 228–241. https://doi.org/10.1145/292540.292561

Andreas Nuyts and Dominique Devriese. 2018. Degrees of Relatedness: A Unified Framework for Parametricity, Irrelevance, Ad Hoc Polymorphism, Intersections, Unions and Algebra in Dependent Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 779–788. https://doi.org/10.1145/3209108.3209119

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

Jens Palsberg and Peter Ørbæk. 1995. Trust in the Lambda-Calculus. In *Proceedings of the Second International Symposium on Static Analysis (SAS '95)*. Springer-Verlag, Berlin, Heidelberg, 314–329.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2013. Coeffects: Unified Static Analysis of Context-Dependence. In *Automata, Languages, and Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 385–397.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A calculus of context-dependent computation. In *Proceedings of International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP 2014)*.

F. Pfenning. 2001. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 221–230. https://doi.org/10.1109/LICS.2001.932499

Andrew M. Pitts. 2013. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, USA.

Frédéric Prost. 2000. A Static Calculus of Dependencies for the $\lambda$-Cube. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS '00)*. IEEE Computer Society, USA, 267.

Naokata Shikuma and Atsushi Igarashi. 2006. Proving Noninterference by a Fully Complete Translation to the Simply Typed $\lambda$-Calculus. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues* (Tokyo, Japan) *(ASIAN'06)*. Springer-Verlag, Berlin, Heidelberg, 301–315.

Geoffrey Smith and Dennis Volpano. 1998. Secure Information Flow in a Multi-Threaded Imperative Language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/268946.268975

Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. 2017. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming* 27 (2017), e5. https://doi.org/10.1017/S0956796816000241

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (Nice, Nice, France) *(TLDI '07)*. Association for Computing Machinery, New York, NY, USA, 53–66. https://doi.org/10.1145/1190315.1190324

Yan Mei Tang and Pierre Jouvelot. 1995. Effect Systems with Subtyping. In *In ACM Conference on Partial Evaluation and Program Manipulation*. ACM Press, 45–53.

Matúš Tejiščák. 2020. A Dependently Typed Calculus with Pattern Matching and Erasure Inference. *Proc. ACM Program. Lang.* 4, ICFP, Article 91 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408973

Frank Tip. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3 (1995), 121–189.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (feb 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Stephen Tse and Steve Zdancewic. 2004. Translating Dependency into Parametricity. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming* (Snow Bird, UT, USA) *(ICFP '04)*. Association for Computing Machinery, New York, NY, USA, 115–125. https://doi.org/10.1145/1016850.1016868

David N. Turner and Philip Wadler. 1999. Operational Interpretations of Linear Logic. *Theor. Comput. Sci.* 227, 1–2 (sep 1999), 231–248. https://doi.org/10.1016/S0304-3975(99)00054-7

Tarmo Uustalu and Varmo Vene. 2008. Comonadic Notions of Computation. *Electronic Notes in Theoretical Computer Science* 203, 5 (2008), 263–284. https://doi.org/10.1016/j.entcs.2008.05.029 Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science (CMCS 2008).

Matthijs Vákár. 2015. Syntax and Semantics of Linear Dependent Types. arXiv:1405.0033 [cs.LO]

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2–3 (jan 1996), 167–187.

Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*. North.

Philip Wadler. 1994. A syntax for linear logic. In *Mathematical Foundations of Programming Semantics*, Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 513–529.

Philip Wadler and Peter Thiemann. 2003. The Marriage of Effects and Monads. *ACM Trans. Comput. Logic* 4, 1 (jan 2003), 1–32. https://doi.org/10.1145/601775.601776

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (aug 2017), 29 pages. https://doi.org/10.1145/3110275