UNIVERSITY OF KENT
THESIS TEMPLATE

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF PHD.

By

Jack Hughes

May 2023

Abstract

A type-directed program synthesis tool can leverage the information provided by resourceful types (linear and graded types) to prune ill-resourced programs from the search space of candidate programs. Therefore, barring any other specification information, a synthesise tool will synthesise a target program (if one exists) more quickly when given a type specification which includes resourceful types than when given an equivalent non-resourceful type.

Acknowledgements

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Chapter 2

Background

This chapter provides the relevant background information. It largely

We explore two lineages of resourceful type systems. In the first, modern resourceful type systems trace their roots to Girard's Linear Logic which was one of the first treatments of data as a resource inside a program. Bounded Linear Logic (BLL) developed this idea further, refining the coarse grained view of data as either linear or non linear. Several subsequent works generalised BLL, coalescing into the notion of a *graded* type system in languages such as Granule . This lineage treats these systems essentially as refinements of an underlying linear structure.

At the same time that these systems were being studied, resourceful types were also being approached from an entirely different perspective — that of computational effects. JOH: more to go here.

Terminology Before delving into linear and graded systems, we briefly frame the approach we will take to discussing the relevant background material. Throughout this chapter (and subsequent chapters) we will tend towards using a *types-and-programs* terminology rather than *propositions-and-proofs*. Through the lens of the Curry-Howard correspondence, one can switch smoothly to viewing our approach to program synthesis as proof search in logic.

The functional programming languages we discuss are presented as typed calculi given by sets of *types*, *terms* (programs), and *typing rules* that relate a term to its type.

Table 1: Timeline.

Year	Linear	Graded
1986		∗ D.K. Gifford, J.M. Lucassen *Integrating Functional and Imperative Programming*
1987	∗ J.Y. Girard *Linear Logic*	
1990		∗ E. Moggi *Notions of Computation and Monads*
1991	∗ J.Y. Girard et al. *Bounded Linear Logic*	
1994	∗ J.S. Hodas, D. Miller *Logic Programming in a Fragment of Intuitionistic Linear Logic*	
2000		∗ P. Wadler, P. Thiemann *Marriage of effects and monads*
2011	∗ U.D. Lago, M. Gaboardi *Linear Dependent Types and Relative Completeness*	
2013		∗ T. Petricek et al. *Coeffects: Unified Static Analysis of Context-Dependence*
2014	∗ D.R. Ghica, A. I. Smith *Bounded Linear Types in a Resource Semiring* ∗ A. Brunel et al. *A Core Quantative Coeffect Calculus*	∗ T. Petricek et al. *Coeffects: a Calculus of Context-Dependent Computation*
2016	∗ M. Gaboardi et al. *Combining Effects and Coeffects via Grading*	∗ C. McBride *I Got Plenty o' Nuttin'*
2017		∗ J.P. Bernardy et al. *Linear Haskell*
2018		∗ R. Atkey *Syntax and Semantics of Quantative Type Theory*
2019	∗ Orchard et al. *Quantitative Program Reasoning with Graded Modal Types*	

The most well-known typed calculus is the simply-typed λ-calculus, which corresponds to intuitionistic logic.

A *judgment* defines the typing relation between a type and a term based on a *context*. In the simple typed λ-calculus, judgments have the form: $\Gamma \vdash t : A$, stating that under some context of *assumptions* Γ the program term t can be assigned the type A. An assumption is a name with an associated type, written $x : A$ and corresponds to an in-scope variable in a program.

A term can be related to a type if we can derive a valid judgment through the application of typing rules. The application of these rules forms a tree structure known as a *typing derivation*.

2.1 Linear and substructural logics

Linear logic was introduces by Girard as a way of being more descriptive about the properties of a derivation in intuitionistic logic. In type systems such as the simply typed λ-calculus, the properties of *weakening*, *contraction*, and *exchange* are assumed implicitly. These are typing rules which are *structural* as they determine how the context may be used rather than being directed by the syntax. Weakening is a rule which allows terms that are not needed in a typing derivation to be discarded. Contraction works as a dual to weakening, allowing an assumption in the context to be used more than once. Finally, exchange allows assumptions in a context to arbitrarily re-ordered.

$$\frac{\Gamma \vdash t : B}{\Gamma, x : A \vdash t : B}\ \textsc{Weakening} \qquad \frac{\Gamma, x : A, y : A \vdash t : B}{\Gamma, x : A \vdash t : B}\ \textsc{Contraction}$$

$$\frac{\Gamma_1, y : B, x : A, \Gamma_2 \vdash t : C}{\Gamma_1, x : A, y : B, \Gamma_2 \vdash t : C}\ \textsc{Exchange}$$

Linear logic is known as a *substructural* logic because it lacks the weakening and contraction rules, while permitting exchange.

The disallowance of these rules means that in order to construct of a typing derivation, each assumption must be used exactly once — arbitrarily copying or discarding values is disallowed, excluding a vast number of programs from being typeable in linear logic. Non-restricted usage of a value is recovered through the modal operator ! (also called “bang”, “of-course”, or the *exponential* modality). Affixing ! to a type captures the notion that values of that type may be used freely in a program.

providing a binary view of data as a resource inside a program: values are either linear or completely unrestricted.

Bounded Linear Logic, took this idea further — instead of a single modal operator, ! is replaced with a family of modal operators indexed by terms which provide an upper bound on usage . These terms provide an upper bound on the usage of a values inside term, e.g. $!_3 A$ is the type of A values which may be used up to 3 times.

JOH: more about Hodas and Miller, ICFP and Granule etc The idea of data as a resource inside a program has been extended further by the proliferation of graded type systems.

2.1.1 A graded linear typing calculus

Having seen the resourceful types from linear logic to graded type systems, we are now in a position to define a full typing calculus, based on the linear λ-calculus extended with a graded modal type. This calculus is equivalent to the core calculus Granule, GrMini. Granule’s full type system is an extension of this graded linear core, with the addition of polymorphism, indexed-types, pattern matching, and the ability to use multiple different graded modalities. We refer to this system as the *linear-base* calculus, reflecting the underlying linear structure of the system.

The types of the linear-base calculus are defined as:

$$A, B ::= A \multimap B \mid \Box_r A \qquad \text{(types)}$$

The type $\Box_r A$ is an indexed family of type operators where r is a *grade* ranging over the elements of a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ parameterising the calculus (where $*$ and $+$ are monotonic with respect to the pre-order $\sqsubseteq$).

The syntax of terms provides the elimination and introduction forms:

$$t ::= x \mid \lambda x.t \mid t_1\, t_2 \mid [t] \mid \textbf{let}\, [x] = t_1 \,\textbf{in}\, t_2 \qquad \text{(terms)}$$

In addition the the terms of the linear λ-calculus, we also have the construct $[t]$ which introduces a graded modal type $\Box_r A$ by 'promoting' a term t to the graded modality, and it's dual $\textbf{let}\, [x] = t_1 \,\textbf{in}\, t_2$ eliminates a graded modal value t_1, binding a graded variable x in scope of t_2. The typing rules provide further understanding of the behaviour of these terms.

Typing judgments are of the form $\Gamma \vdash t : A$, where Γ ranges over contexts:

$$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x :_r A \qquad \text{(contexts)}$$

Thus, a context may be empty $\emptyset$, extended with a linear assumption $x : A$ or extended with a graded assumption $x :_r A$. For linear assumptions, structural rules of weakening and contraction are disallowed. Graded assumptions may be used non-linearly according to the constraints given by their grade, the semiring element r. Throughout, comma denotes disjoint context concatenation.

Various operations on contexts are used to capture non-linear data flow via grading. Firstly, *context addition* (2.2.1) provides an analogue to contraction, combining contexts that have come from typing multiple subterms in a rule. Context addition, written $\Gamma_1 + \Gamma_2$, is undefined if Γ_1 and Γ_2 overlap in their linear assumptions. Otherwise graded assumptions appearing in both contexts are combined via the semiring $+$ of their grades.

JOH: also provide declarative specification

Definition 2.1.1 (Context addition)**.**

$$\begin{array}{rll}
(\Gamma, x : A) + \Gamma' = (\Gamma + \Gamma'), x : A & \text{iff } x \notin |\Gamma'| & \emptyset + \Gamma = \Gamma \\
\Gamma + (\Gamma', x : A) = (\Gamma + \Gamma'), x : A & \text{iff } x \notin |\Gamma| & \Gamma + \emptyset = \Gamma \\
(\Gamma, x :_r A) + (\Gamma', x :_s A) = (\Gamma + \Gamma'), x :_{(r+s)} A & &
\end{array}$$

Note that this is a declarative specification of context addition. Graded assumptions may appear in any position in Γ and Γ' as witnessed by the algorithmic specification where for all Γ_1, Γ_2 *context addition* is defined as follows by ordered cases matching inductively on the structure of Γ_2:

$$\Gamma_1 + \Gamma_2 = \begin{cases} \Gamma_1 & \Gamma_2 = \emptyset \\ ((\Gamma_1', \Gamma_1'') + \Gamma_2'), x :_{(r+s)} A & \Gamma_2 = \Gamma_2', x :_s A \wedge \Gamma_1 = \Gamma_1', x :_r A, \Gamma_1'' \\ (\Gamma_1 + \Gamma_2'), x : A & \Gamma_2 = \Gamma_2', x : A \;\wedge\; x : A \notin \Gamma_1 \end{cases}$$

$$\frac{}{x : A \vdash x : A}\textsc{Var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\textsc{Abs} \qquad \frac{\Gamma_1 \vdash t_1 : A \to B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\textsc{App}$$

$$\frac{\Gamma \vdash t : A}{\Gamma, [\Delta]_0 \vdash t : A}\textsc{Weak} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x :_1 A \vdash t : B}\textsc{Der} \qquad \frac{\Gamma, x :_r A, \Gamma' \vdash t : A \quad r \sqsubseteq s}{\Gamma, x :_s A, \Gamma' \vdash t : A}\textsc{Approx}$$

$$\frac{[\Gamma] \vdash t : A}{r \cdot [\Gamma] \vdash [t] : \Box_r A}\textsc{Pr} \qquad \frac{\Gamma_1 \vdash t_1 : \Box_r A \quad \Gamma_2, x :_r A \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\, [x] = t_1\, \textbf{in}\, t_2 : B}\textsc{Let}\Box$$

Figure 1: Typing rules of the graded linear λ-calculus

Figure 2 defines the typing rules. Linear variables are typed in a singleton context (Var). Abstraction (Abs) and application (App) follow the rules of the linear λ-calculus. The Weak rule captures weakening of assumptions graded by 0 (where $[\Delta]_0$ denotes a context containing only graded assumptions graded by 0). Context addition and Weak together therefore provide the rules of substructural rules of contraction and weakening. Dereliction (Der), allows a linear assumption to be converted to a graded assumption with grade 1. Grade approximation is captured by the Approx rule, which allows a grade s to be converted to another grade r, providing that r *approximates* s, where the relation $\sqsubseteq$ is the pre-order provided with the semiring. Introduction and elimination of the graded modality is provided by the Pr and Let rules respectively. The Pr rule propagates the grade r to the assumptions through *scalar multiplication* of Γ by r where every assumption in Γ must already be graded (written $[\Gamma]$ in the rule), given by definition (2.1.2).

Definition 2.1.2 (Scalar context multiplication)**.** A context which consists solely of graded assumptions can be multiplied by a semiring grade $r \in \mathcal{R}$

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

The Let rule eliminates a graded modal value $\Box_r A$ into a graded assumption $x :_r A$ with a matching grade in the scope of the **let** body. This is also referred to as “unboxing”.

We give an example of graded modalities using a graded modality indexed by the semiring of natural numbers.

Example 2.1.1. The natural number semiring with discrete ordering $(\mathbb{N}, *, 1, +, 0, \equiv)$ provides a graded modality that counts exactly how many times non-linear values are used. As a simple example, the S combinator from the SKI system of combinatory logic is typed and defined:

$$\begin{aligned} &s : (A \to (B \to C)) \to (A \to B) \to (\Box_2 A \to C) \\ &s = \lambda x.\lambda y.\lambda z'.\mathbf{let}\,[z] = z'\,\mathbf{in}\,(x\,z)\,(y\,z) \end{aligned}$$

The graded modal value z' captures the ‘capability’ for a value of type A to be used twice. This capability is made available by eliminating $\Box$ (via **let**) to the variable z, which is graded $z : [A]_2$ in the scope of the body.

Metatheory The admissibility of substitution is a key result that holds for this language **?**, which is leveraged in soundness of the synthesis calculi.

Lemma 2.1.1 (Admissibility of substitution)**.** *Let* $\Delta \vdash t' : A$*, then:*

- *(Linear) If* $\Gamma, x : A, , \Gamma' \vdash t : B$ *then* $\Gamma + \Delta + \Gamma' \vdash [t'/x]t : B$
- *(Graded) If* $\Gamma, x :_r A, , \Gamma' \vdash t : B$ *then* $\Gamma + (r \cdot \Delta) + \Gamma' \vdash [t'/x]t : B$

2.2 Graded types: An alternative history

JOH: talk about lineage of graded type systems

2.2.1 Graded-base

We now define a core calculus for a fully graded type system. This system is much closer to those outlined above than the linear-base calculus, drawing from the coeffect calculus of **?**, Quantitative Type Theory (QTT) by **?** and refined further by **?** (although we omit dependent types from our language), the calculus of **?**, and other graded dependent type theories **??**. Similar systems also form the basis of the core of the linear types extension to Haskell **?**. We refer to this system as the *graded-base* calculus to differentiate it from linear-base.

The syntax of graded-base types is given by:

$$A, B ::= A^r \to B \mid \Box_r A \qquad (\textit{types})$$

where the function space $A^r \to B$ annotates the input type with a *grade* r drawn from a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ which paramterising the calculus as in linear-base.

The graded necessity modality $\Box_r A$ is similarly annotated by the grade r being an element of the semiring.

The syntax of terms is given as:

$$t ::= x \mid \lambda x.t \mid t_1\, t_2 \mid [t] \qquad (\textit{terms})$$

Similarly to linear-base, terms consist of a graded λ-calculus, extended with a *promotion* construct [t] which introduces a graded modality explicitly.

Definition 2.2.1 (Context addition)**.**

$$\Gamma_1 + \Gamma_2 = \begin{cases} \Gamma_1 & \Gamma_2 = \emptyset \\ ((\Gamma_1', \Gamma_1'') + \Gamma_2'), x :_{(r+s)} A & \Gamma_2 = \Gamma_2', x :_s A \wedge \Gamma_1 = \Gamma_1', x :_r A, \Gamma_1'' \\ (\Gamma_1 + \Gamma_2'), x :_s A & \Gamma_2 = \Gamma_2', x :_s A \wedge x \notin \mathsf{dom}(\Gamma_1) \end{cases}$$

$$\frac{}{0 \cdot \Gamma, x :_1 A \vdash x : A}\ \textsc{TyVar} \qquad \frac{\Gamma, x :_r A \vdash t : B}{\Gamma \vdash \lambda x.t : A^r \to B}\ \textsc{TyAbs} \qquad \frac{\Gamma_1 \vdash t_1 : A^r \to B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + r \cdot \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{TyAp}$$

$$\frac{\Gamma \vdash t : A}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{TyPr} \qquad \frac{\Gamma, x :_r A, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x :_s A, \Gamma' \vdash t : B}\ \textsc{TyApprox}$$

Figure 2: Typing rules for graded-base

Figure 2 gives the full typing rules, which helps explain the meaning of the syntax with reference to their static semantics.

Variables (rule Var) are typed in a context where the variable x has grade 1 denoting its single usage here. All other variable assumptions are given the grade of the 0 semiring element (providing *weakening*), using *scalar multiplication* of contexts by a grade, re-using definition (2.1.2).

2.3 Two typed calculi

Having outlined the two major lineages of resourceful types, we are now left with the question: what approach should we use as the basis of our program synthesis tool? Both approaches pose there own challenges and questions which influence the design of a synthesis calculus. The breadth of these differences leaves omitting either approach an undesirable prospect. For this reason, this thesis tackles languages based on both approaches:

1. We begin with a simplified synthesis calculus for a core language based on the linear λ calculus outlined in section (2.1.1). To better illustrate the practicality of the syntesis calculus, the language is extended with product and sum types, as well as a unit type.

2.

This chapter introduced some of the key features of languages with resourceful types.

Linear and graded types embed usage constraints in the typing rules, enforcing the notion that a well-typed program is also *well-resourced*.

The next chapter focuses on the linear-base core calculus of section 2.1.1, extending this calculus with multiplicative and additive types, as well as a unit type to form a more practical programming language. This then comprises the target language of a synthesis algorithm. Likewise, the graded-base calculus is revisited in chapter **??**, where it is extended with (G)ADTs, and recursion, providing a target language for a more in-depth and featureful synthesis tool.

Chapter 3

A core synthesis calculus

Chapter 4

Deriving graded combinators

Chapter 5

An extended synthesis calculus

Chapter 6

Conclusion