UNIVERSITY OF KENT
THESIS TEMPLATE

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF PHD.

Draft on May 22, 2023

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

JOH: Try to outline the approach of the background - the focus is on introducing the two languages: graded and linear base. Why do we need two languages? What is the difference between them? What do we use each language for in synthesis - i.e. why do we decide to do the initial calculus in the linear base and then shift to the graded base for the more advanced stuff? This chapter provides the background material necessary for understanding the design of synthesis calculi. This is done primarily by formally introducing two programming languages which include resourceful types. Language features, and concepts, are explained as they arise. We begin with an outline of linear and graded types themselves, briefly charting their origins from linear logic and bounded linear logic. Following this, two languages are presented which make use of these resourceful types in different ways. These two languages will become the bases of the target languages of two synthesis calculi in subsequent chapters.

JOH: How much detail should be provided here? It's very likely that we will cover a brief history of graded type systems (from linear logic to graded modal types) in the intro. So should we use this space to go into further detail? Or cut it here and leave it in the intro

Linear types

Graded types JOH: possibly doesn't need to be its own section

2.1 Two target languages

When designing a type-directed synthesis tool, a natural first consideration would be the *target* language of synthesis: what programming language will synthesised programs be written in? What types will be used to specify a desired program? If we strip away all but the most essential language elements that we wish to include in our target language, a good starting point would be the simply typed linear λ-calculus, extended with a graded modal type. The linear λ-calculus has the same syntax as the standard simply typed λ calculus only with the structural rules of weakening and contraction prohibited, making terms which express non-linear behaviour ill-typed.

This calculus is equivalent to the core calculus Granule, GrMini. Granule's full type system is an extension of this graded linear core, introducing (G)ADTs and recurison.

For this chapter we concern ourselves only with the λ-calculus core. In subsequent chapters these calculi will be extended further to become more full-fledged programming languages.JOH: How?

An alternative approach is to replace this underlying linear system with a *graded* λ-calculus.

In this system, function types are augmented with a grade: $A^r \to B$ which desrcibes how the value of the argument type may be used in the body of the function. This approach more closely resembles the graded systems of JOH: list some systems here, linear haskell, QTT, etc..

The different approaches to graded types here gives rise to the question: how might the features of the target language influence the design of a synthesis tool?

JOH: It's still not clear why we choose these two languages - or why we choose two languages at all instead of focusing on one, possibly need to come up with a better justification

The two approaches are now outlined formally in turn, beginning with the linear-base calculus.

2.1.1 Linear-base

The types of the graded linear λ-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:

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

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 $\mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2$ 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

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

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

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

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

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* 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.

Definition 2.1.1 (Context addition)**.** For all Γ_1, Γ_2 *context addition* is defined as follows by ordered cases matching inductively on the structure of Γ_2:

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

JOH: Possibly some example programs in this section? The problem might be that they'll have to be purely lambda calc programs

Figure 1 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). 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), defined:

Definition 2.1.2 (Scalar context multiplication)**.**

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

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.

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 $\Gamma + (r \cdot \Delta) + \Gamma' \vdash [t'/x]t : B$*

2.1.2 Graded-base

We now define our second core language with graded types. This system will be familiar to those who have encountered graded types before, 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 **?**. This calculus shares much in common with languages based on linear types, such as the graded monadic-comonadic calculus of **?**, generalisations of Bounded Linear Logic **??**

This system corresponds to Granule's "graded base" language extension rather than its default linear types basis outlined above.

Again, our system is parameterised by a graded necessity modality. The syntax of 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 $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ parameterising the calculus. Type constructors K include the unit and multiplicative linear product type but are also extended by user-defined ADTs in the implementation which can be formed by products, coproducts, and recursive types (although we do not given an explicit syntax to the latter two type formers here). The graded necessity modality $\Box_r A$ is similarly annotated by the grade r being an element of the semiring.

JOH: Do we want to include pattern matching here? In fact, do we want to include it in the linear-base calculus from the start as well? The syntax of terms is given as:

$$t ::= x \mid \lambda x.t \mid t_1\ t_2 \mid [t] \mid C\ t_1 \ldots t_n \mid \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; \ldots; p_n \mapsto t_n \qquad (\textit{terms})$$

$$p ::= x \mid \_ \mid [p] \mid C\ p_1 \ldots p_n \qquad (\textit{patterns})$$

Terms consist of a graded λ-calculus, a *promotion* construct [t] which introduces a graded modality explicitly, as well as data constructor introduction ($C\ t_1 \ldots t_n$) and elimination via **case** expressions which are defined via the syntax of patterns

p.

JOH: Talk about how we overload the definition from linear-base but removing the linear assumption case

Definition 2.1.3 (Context addition)**.** For all Γ_1, Γ_2 *context addition* is defined as follows by ordered cases matching inductively on the structure of Γ_2:

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

JOH: No need to redefine scalar context multiplication

2.2 Related systems

2.3 Conclusion

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