UNIVERSITY OF KENT

PROGRAM SYNTHESIS FROM LINEAR AND GRADED TYPES

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF PHD.

By

Jack Hughes

August 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

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*	

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. 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* (5.1.2) 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)**.**

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

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 6 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 (5.1.1).

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:

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

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)**.**

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

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

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

Figure 2: Typing rules for graded-base

Figure 6 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 (5.1.1).

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

3.1 A Practical Target Language

Our linear base calculus presented in chapter 2 contains only the absolute essential core of a functional programming language with graded modalities. We extend the syntax with multiplicatve product types $\otimes$, additive coproduct types $\oplus$, and a multiplicative unit 1. The syntax for these extensions is given by the following grammar, which extends the linear-base syntax of section 2.1.1:

$$\begin{aligned} t ::= {} & x \mid \lambda x.t \mid t_1\, t_2 \mid [t] \mid \textbf{let}\, [x] = t_1 \,\textbf{in}\, t_2 \mid (t_1, t_2) \mid \textbf{let}\, (x_1, x_2) = t_1 \,\textbf{in}\, t_2 \\ & \mid () \mid \textbf{let}\, () = t_1 \,\textbf{in}\, t_2 \mid \textbf{inl}\, t \mid \textbf{inr}\, t \mid \textbf{case}\ t_1\ \textbf{of inl}\ x_1 \to t_2;\ \textbf{inr}\ x_2 \to t_3 \end{aligned} \qquad \text{(terms)}$$

We use the syntax () for the inhabitant of the multiplicative unit 1. Pattern matching via a **let** is used to eliminate products and unit types; for sum types, **case** is used to distinguish the constructors.

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

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \quad \Gamma_2, x_1 : A, x_2 : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x_1, x_2) = t_1\,\mathbf{in}\,t_2 : C}\ \textsc{LetPair}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathbf{inl}\,t : A \oplus B}\ \textsc{Inl} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathbf{inr}\,t : A \oplus B}\ \textsc{Inr}$$

$$\frac{\Gamma_1 \vdash t_1 : A \oplus B \qquad \Gamma_2, x_1 : A \vdash t_2 : C \qquad \Gamma_3, x_2 : B \vdash t_3 : C}{\Gamma + (\Gamma_2 \sqcup \Gamma_3) \vdash\ \mathbf{case}\ t_1\ \mathbf{of\ inl}\ x_1 \rightarrow t_2;\ \mathbf{inr}\ x_2 \rightarrow t_3 : C}\ \textsc{Case}$$

$$\frac{}{\emptyset \vdash () : \mathbf{1}}\ 1 \qquad \frac{\Gamma_1 \vdash t_1 : \mathbf{1} \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : A}\ \textsc{Let1}$$

Figure 3: Typing rules of for $\otimes$, $\oplus$, and 1

Figure 3 gives the typing rules. Rules for multiplicative products (pairs) and additive coproducts (sums) are routine, where pair introduction (Pair) adds the contexts used to type the pair's constituent subterms. Pair elimination (LetPair) binds a pair's components to two linear variables in the scope of the body t_2. The Inl and Inr rules handle the typing of constructors for the sum type $A \oplus B$. Elimination of sums (Case) takes the least upper bound (defined above) of the contexts used to type the two branches of the case.

In the typing of **case** expressions, the *least-upper bound* of the two contexts used to type each branch is used, defined:

Definition 3.1.1 (Partial least-upper bounds of contexts)**.** For all Γ_1, Γ_2:

$$\Gamma_1 \sqcup \Gamma_2 = \begin{cases} \emptyset & \Gamma_1 = \emptyset & \wedge\, \Gamma_2 = \emptyset \\ (\emptyset \sqcup \Gamma_2'), x :_{0 \sqcup s} A & \Gamma_1 = \emptyset & \wedge\, \Gamma_2 = \Gamma_2', x :_s A \\ (\Gamma_1' \sqcup (\Gamma_2', \Gamma_2'')), x : A & \Gamma_1 = \Gamma_1', x : A & \wedge\, \Gamma_2 = \Gamma_2', x : A,\ , \Gamma_2'' \\ (\Gamma_1' \sqcup (\Gamma_2', \Gamma_2'')), x :_{r \sqcup s} A & \Gamma_1 = \Gamma_1', x :_r A & \wedge\, \Gamma_2 = \Gamma_2', x :_s A, \Gamma_2'' \end{cases}$$

where $r \sqcup s$ is the least-upper bound of grades r and s if it exists, derived from $\sqsubseteq$.

As an example of the partiality of $\sqcup$, if one branch of a **case** uses a linear variable, then the other branch must also use it to maintain linearity overall, otherwise the upper-bound of the two contexts for these branches is not defined.

With these extensions in place, we now have the capacity to write more idiomatic functional programs in our target language. As a demonstration of this, and to showcase how graded modalities interact with these new type extensions, we provide two further examples of different graded modalities which complement these new types.

Example 3.1.1. Exact usage analysis is less useful when control-flow is involved, e.g., eliminating sum types where each control-flow branch uses variables differently. The above $\mathbb{N}$-semiring can be imbued with a notion of *approximation* via less-than-equal ordering, providing upper bounds. A more expressive semiring is that of natural number intervals **?**, given by pairs $\mathbb{N} \times \mathbb{N}$ written $[r...s]$ here for the lower-bound $r \in \mathbb{N}$ and upper-bound usage $s \in \mathbb{N}$ with $0 = [0...0]$ and $1 = [1...1]$, addition and multiplication defined pointwise, and ordering $[r...s] \sqsubseteq [r'...s'] = r' \leq r \wedge s \leq s'$. Thus a coproduct elimination function can be written and typed:

$$\oplus_e : \Box_{[0...1]}(A \multimap C) \multimap \Box_{[0...1]}(B \multimap C) \multimap (A \oplus B) \multimap C$$

$$\oplus_e = \lambda x'.\lambda y'.\lambda z.\textbf{let } [x] = x' \textbf{ in let } [y] = y' \textbf{ in case } z \textbf{ of inl } u \rightarrow x\ u \mid \textbf{inr } v\ \rightarrow y\ v$$

Example 3.1.2. Graded modalities can capture a form of information-flow security, tracking the flow of labelled data through a program **?**, with a lattice-based semiring on $\mathcal{R} = \{\mathsf{Unused} \sqsubseteq \mathsf{Hi} \sqsubseteq \mathsf{Lo}\}$ where $0 = \mathsf{Unused}$, $1 = \mathsf{Hi}$, $+ = \sqcup$ and if $r = \mathsf{Unused}$ or $s = \mathsf{Unused}$ then $r \cdot s = \mathsf{Unused}$ otherwise $r \cdot s = \sqcup$. This allows the following well-typed program, eliminating a pair of Lo and Hi security values, picking the left one to pass to a continuation expecting a Lo input:

$$\begin{aligned}
&\mathit{noLeak} : (\Box_{\mathsf{Lo}} A \otimes \Box_{\mathsf{Hi}} A) \rightarrow (\Box_{\mathsf{Lo}}(A \otimes 1) \rightarrow B) \rightarrow B \\
&\mathit{noLeak} = \lambda z.\lambda z'.\mathbf{let}\,(x', y') = z\,\mathbf{in}\,\mathbf{let}\,[x] = x'\,\mathbf{in}\,\mathbf{let}\,[y] = y'\,\mathbf{in}\,z'\,[(x, ())]
\end{aligned}$$

3.2 The Resource Management Problem

Chapter **??** discussed a rule for synthesising pairs and highlighted how graded types could be use to control the number of times assumptions are used in the synthesising term. In a linear or graded context, synthesis needs to handle the problem of *resource management* **??**: how do we give a resourceful accounting to the context during synthesis so that we respect its constraints. Before explicating our synthesis approach, we give an overview of the resource management problem here.

Chapter **??** considered (Cartesian) product types $\times$, but in our target language we switch to the *multiplicative product* of linear types, given in figure 3Each subterm is typed by a different context Γ_1 and Γ_2 which are then combined via *disjoint* union: the pair cannot be formed if variables are shared between Γ_1 and Γ_2. This prevents the structural behaviour of *contraction* (where a variable appears in multiple subterms). Naïvely inverting this typing rule into a synthesis rule yields:

$$\frac{\Gamma_1 \vdash A \Rightarrow t_1 \qquad \Gamma_2 \vdash B \Rightarrow t_2}{\Gamma_1, \Gamma_2 \vdash A \otimes B \Rightarrow (t_1, t_2)}\ \otimes_{\textsc{Intro}}$$

As a declarative specification, the $\otimes_{\textsc{Intro}}$ synthesis rule is sufficient. However, this rule embeds a considerable amount of non-determinism when considered from an algorithmic perspective. Reading 'clockwise' starting from the bottom-left, given some context Γ and a goal $A \otimes B$, we have to split the context into disjoint subparts Γ_1 and Γ_2 such that $\Gamma = \Gamma_1, \Gamma_2$ in order to pass the Γ_1 and Γ_2 to the subgoals for A and B. For a context of size n there are 2^n possible such partitions! This quickly becomes intractable. Instead, Hodas and Miller developed a technique for linear logic programming **?**, refined by Cervasto et al. **?**, where proof search for linear logic has both an *input context* of available resources and an *output context* of the remaining resources, which we write as judgements of the form $\Gamma \vdash A \Rightarrow^{-} t \mid \Gamma'$ for input context Γ and output context Γ'. Synthesis for multiplicative products then becomes:

$$\frac{\Gamma_1 \vdash A \Rightarrow^{-} t_1 \mid \Gamma_2 \qquad \Gamma_2 \vdash B \Rightarrow^{-} t_2 \mid \Gamma_3}{\Gamma_1 \vdash A \otimes B \Rightarrow^{-} (t_1, t_2) \mid \Gamma_3}\ \otimes^{-}_{\textsc{Intro}}$$

where the remaining resources after synthesising for A the first term t_1 are Γ_2 which are then passed as the resources for synthesising the second term B. There is an ordering implicit here in 'threading through' the contexts between the premises. For example, starting with a context $x : A, y : B$, then this rule can be instantiated as:

$$\frac{x : A, y : B \vdash A \Rightarrow^{-} x \mid y : B \qquad y : B \vdash B \Rightarrow^{-} y \mid \emptyset}{x : A, y : B \vdash A \otimes B \Rightarrow^{-} (x, y) \mid \emptyset}\ \otimes^{-}_{\textsc{Intro}} \qquad \text{(example)}$$

Thus this approach neatly avoids the problem of having to split the input context, and facilitates efficient proof search for linear types. This idea was adapted by Hughes and Orchard to graded types to facilitate the synthesis of programs in Granule **?**. Hughes and Orchard termed the above approach *subtractive* resource management (in a style similar to *left-over* type-checking for linear type systems **??**). In a graded setting however, this approach was shown to be costly.

Graded type systems, as we consider them here, have typing contexts in which free-variables are assigned a type, and a grade (usually drawn from some semiring structure parameterising the calculus). A useful example is the semiring of natural numbers which is used to describe exactly how many times an assumption can be used (in contrast to linear assumptions which must be used exactly once). For example, the context $x :_2 A, y :_0 B$ explains that x must be used twice but y must be used not at all. The literature contains many other examples of semirings for tracking other properties in this way, such as security labels **???**, intervals of usage **?**, or hardware schedules **?**. In a graded setting, the subtractive approach is problematic as there is not necessarily a notion of actual subtraction for grades. Consider a version of the above example for subtractively synthesising a pair, but now for a context with some grades r and s on the input variables. Using a variable to synthesise a subterm now does not result in that variable being left out of the output context. Instead a new grade must be assigned in the output context that relates to the first by means of an additional constraint describing that some usage took place:

$$\frac{\begin{array}{c}\exists r'.r' + 1 = r \qquad \exists s'.s' + 1 = s \qquad x :_r A, y :_s B \vdash A \Rightarrow^- x \mid x :_{r'} A, y :_s B \\ x :_{r'} A, y :_s B \vdash B \Rightarrow^- y \mid x :_{r'} A, y :_{s'} B\end{array}}{x :_r A, y :_s B \vdash A \otimes B \Rightarrow^- (x, y) \mid x :_{r'} A, y :_{s'} B}\ \otimes^-_{\textsc{Intro}} \qquad \text{(example)}$$

In the first synthesis premise, x has grade r in the input context, x is synthesised for the goal, and thus the output context has some grade r' where $r' + 1 = r$, denoting that some usage of x occurred (which is represented by the 1 element of the semiring in graded systems).

For the natural numbers semiring, with $r = 1$ and $s = 1$ then the constraints above are satisfied with $r' = 0$ and $s' = 0$. In a general setting, this subtractive approach to synthesis for graded types requires solving many such existential equations over semirings, which also introduces a new source of non-determinism is there is more than one solution.

3.3 Synthesis calculi

We present two linear-base synthesis calculi with subtractive and additive resource management schemes, extending an input-output context management approach to graded modal types. The structure of the synthesis calculi mirrors a cut-free sequent calculus, with *left* and *right* rules for each type constructor. Right rules synthesise an introduction form for the goal type. Left rules eliminate (deconstruct) assumptions so that they may be used inductively to synthesise subterms. Each type in the core language has right and left rules corresponding to its constructors and destructors respectively.

3.3.1 Subtractive Resource Management

Our subtractive approach follows the philosophy of earlier work on linear logic proof search **??**, structuring synthesis rules around an input context of the available resources and an output context of the remaining resources that can be used to synthesise subsequent subterms. Synthesis rules are read bottom-up, with judgments $\Gamma \vdash A \Rightarrow^{-} t \mid \Delta$ meaning from the *goal type* A we can synthesise a term t using assumptions in Γ, with output context Δ. We describe the rules in turn to aid understanding. Figure **??** collects the rules for reference:

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^{-} x \mid \Gamma}\ \textsc{LinVar}^{-} \qquad \frac{\exists s.\, r \sqsubseteq s + 1}{\Gamma, x :_r A \vdash A \Rightarrow^{-} x \mid \Gamma, x :_s A}\ \textsc{GrVar}^{-}$$

$$\frac{\Gamma, x : A \vdash B \Rightarrow^{-} t \mid \Delta \qquad x \notin |\Delta|}{\Gamma \vdash A \multimap B \Rightarrow^{-} \lambda x.t \mid \Delta}\ {\multimap}_R^{-}$$

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1 \vdash A \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{-} [(x_1\, t_2)/x_2]t_1 \mid \Delta_2}\ {\multimap}_L^{-}$$

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^{-} t \mid \Delta, x :_{s'} A \qquad y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s + 1}{\Gamma, x :_r A \vdash B \Rightarrow^{-} [x/y]t \mid \Delta, x :_{s'} A}\ \textsc{der}^{-}$$

$$\frac{\Gamma \vdash A \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^{-} [t] \mid \Gamma - r \cdot (\Gamma - \Delta)}\ \Box_R^{-} \qquad \frac{\Gamma, x_2 :_r A \vdash B \Rightarrow^{-} t \mid \Delta, x_2 :_s A \qquad 0 \sqsubseteq s}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^{-} \textbf{let}\, [x_2] = x_1\, \textbf{in}\, t \mid \Delta}\ \Box_L^{-}$$

$$\frac{\Gamma \vdash A \Rightarrow^{-} t_1 \mid \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{-} (t_1, t_2) \mid \Delta_2}\ \otimes_R^{-}$$

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^{-} t_2 \mid \Delta \qquad x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^{-} \textbf{let}\, (x_1, x_2) = x_3\, \textbf{in}\, t_2 \mid \Delta}\ \otimes_L^{-}$$

$$\frac{\Gamma \vdash A \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{-} \textbf{inl}\, t \mid \Delta}\ \oplus 1_R^{-} \qquad \frac{\Gamma \vdash B \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{-} \textbf{inr}\, t \mid \Delta}\ \oplus 2_R^{-}$$

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \quad \Gamma, x_3 : B \vdash C \Rightarrow^{-} t_2 \mid \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2|}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^{-} \textbf{case}\ x_1\ \textbf{of}\ \textbf{inl}\ x_2 \to t_1;\ \textbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcap \Delta_2}\ \oplus_L^{-}$$

$$\frac{}{\Gamma \vdash 1 \Rightarrow^{-} () \mid \Gamma}\ 1_R^{-} \qquad \frac{\Gamma \vdash C \Rightarrow^{-} t \mid \Delta}{\Gamma, x : 1 \vdash C \Rightarrow^{-} \textbf{let}\, () = x\, \textbf{in}\, t \mid \Delta}\ 1_L^{-}$$

Figure 4: Collected rules of the subtractive synthesis calculus

Variables Variable terms can be synthesised from linear or graded assumptions by rules:

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^{-} x \mid \Gamma}\textsc{LinVar}^{-} \qquad \frac{\exists s.\, r \sqsubseteq s + 1}{\Gamma, x :_r A \vdash A \Rightarrow^{-} x \mid \Gamma, x :_s A}\textsc{GrVar}^{-}$$

On the left, a variable x may be synthesised for the goal A if a linear assumption $x : A$ is present in the input context. The input context without x is then returned as the output context, since x has been used. On the right, we can synthesise a variable x for A we have a graded assumption of x matching the type. However, the grading r must permit x to be used once here. Therefore, the premise states that there exists some grade s such that grade r approximates $s + 1$. The grade s represents the use of x in the rest of the synthesised term, and thus $x :_s A$ is in the output context. For the natural numbers semiring, this constraint is satisfied by $s = r - 1$ whenever $r \neq 0$, e.g., if $r = 3$ then $s = 2$. For intervals, the role of approximation is more apparent: if $r = [0...3]$ then this rule is satisfied by $s = [0...2]$ where $s + 1 = [0...2] + [1...1] = [1...3] \sqsubseteq [0...3]$. This is captured by the instantiation of a new existential variable representing the new grade for x in the output context of the rule. In the natural numbers semiring, this could be done by simply subtracting 1 from the assumption's existing grade r. However, as not all semirings have an additive inverse, this is instead handled via a constraint on the new grade s, requiring that $r \sqsupseteq s + 1$. In the implementation, the constraint is discharged via an SMT solver, where an unsatisfiable result terminates this branch of synthesis.

Functions In typing, λ-abstraction binds linear variables to introduce linear functions. Synthesis from a linear function type therefore mirrors typing:

$$\frac{\Gamma, x : A \vdash B \Rightarrow^{-} t \mid \Delta \qquad x \notin |\Delta|}{\Gamma \vdash A \multimap B \Rightarrow^{-} \lambda x.t \mid \Delta} \multimap_R^{-}$$

Thus, $\lambda x.t$ can be synthesised given that t can be synthesised from B in the context of Γ extended with a fresh linear assumption $x : A$. To ensure that x is used linearly by t we must therefore check that it is not present in Δ.

The left-rule for linear function types then synthesises applications (as in **?**):

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1 \vdash A \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{-} [(x_1\, t_2)/x_2]t_1 \mid \Delta_2} \multimap_L^{-}$$

The rule synthesises a term for type C in a context that contains an assumption $x_1 : A \multimap B$. The first premise synthesises a term t_1 for C under the context extended with a fresh linear assumption $x_2 : B$, i.e., assuming the result of x_1. This produces an output context Δ_1 that must not contain x_2, i.e., x_2 is used by t_1. The remaining assumptions Δ_1 provide the input context to synthesise t_2 of type A: the argument to the function x_1. In the conclusion, the application $x_1\, t_2$ is substituted for x_2 inside t_1, and Δ_2 is the output context.

Dereliction Note that the above rule synthesises the application of a function given by a linear assumption. What if we have a graded assumption of function type? Rather than duplicating every left rule for both linear and graded assumptions, we mirror the dereliction typing rule (converting a linear assumption to graded) as:

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^- t \mid \Delta, x :_{s'} A \qquad y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s + 1}{\Gamma, x :_r A \vdash B \Rightarrow^- [x/y]t \mid \Delta, x :_{s'} A}\ \textsc{der}^-$$

Dereliction captures the ability to reuse a graded assumption being considered in a left rule. A fresh linear assumption y is generated that represents the graded assumption's use in a left rule, and must be used linearly in the subsequent synthesis of t. The output context of this premise then contains x graded by s', which reflects how x was used in the synthesis of t, i.e. if x was not used then $s' = s$. The premise $\exists s.\, r \sqsupseteq s + 1$ constrains the number of times dereliction can be applied so that it does not exceed x's original grade r.

Graded modalities For a graded modal goal type $\Box_r A$, we synthesise a promotion $[t]$ if we can synthesise the 'unpromoted' t from A:

$$\frac{\Gamma \vdash A \Rightarrow^- t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^- [t] \mid \Gamma - r \cdot (\Gamma - \Delta)}\ \Box_R^-$$

A non-graded value t may be promoted to a graded value using the box syntactic construct. Recall that typing of a promotion $[t]$ scales all the graded assumptions used to type t by r. Therefore, to compute the output context we must "subtract" r-times the use of the variables in t. However, in the subtractive model Δ tells us what is left, rather than what is used. Thus we first compute the *context subtraction* of Γ and Δ yielding the variables usage information about t:

Definition 3.3.1 (Context subtraction)**.** For all Γ_1, Γ_2 where $\Gamma_2 \subseteq \Gamma_1$:

$$\Gamma_1 - \Gamma_2 = \begin{cases} \Gamma_1 & \Gamma_2 = \emptyset \\ (\Gamma_1', \Gamma_1'') - \Gamma_2' & \Gamma_2 = \Gamma_2', x : A \quad \wedge\ \Gamma_1 = \Gamma_1', x : A, \Gamma_1'' \\ ((\Gamma_1', \Gamma_1'') - \Gamma_2'), x :_q A & \Gamma_2 = \Gamma_2', x :_s A \quad \wedge\ \Gamma_1 = \Gamma_1', x :_r A, \Gamma_1'' \\ & \wedge\ \exists q.\, r \sqsupseteq q + s\ \wedge \forall q'.r \sqsupseteq q' + s \implies q \sqsupseteq q' \end{cases}$$

As in graded variable synthesis, context subtraction existentially quantifies a variable q to express the relationship between grades on the right being "subtracted" from those on the left. The last conjunct states q is the greatest element (wrt. to the pre-order) satisfying this constraint, i.e., for all other $q' \in \mathcal{R}$ satisfying the subtraction constraint then $q \sqsupseteq q'$ e.g., if $r = [2...3]$ and $s = [0...1]$ then $q = [2...2]$ instead of, say, $[0...1]$. This *maximality* condition is important for soundness (that synthesised programs are well-typed).

Thus for $\Box_R^-$, $\Gamma - \Delta$ is multiplied by the goal type grade r to obtain how these variables are used in t after promotion. This is then subtracted from the original input context Γ giving an output context containing the left-over variables and grades. Context multiplication requires that $\Gamma - \Delta$ contains only graded variables, preventing the incorrect use of linear variables from Γ in t.

Synthesis of graded modality elimination, is handled by the $\Box_L^-$ left rule:

$$\frac{\Gamma, x_2 :_r A \vdash B \Rightarrow^- t \mid \Delta, x_2 :_s A \qquad 0 \sqsubseteq s}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^- \textbf{let}\, [x_2] = x_1\, \textbf{in}\, t \mid \Delta}\ \Box_L^-$$

Given an input context comprising Γ and a linear assumption x_1 of graded modal type, we can synthesise an unboxing of x_1 if we can synthesise a term t under Γ extended with a graded assumption $x_2 :_r A$. This returns an output context that must contain x_2 graded by s with the constraint that s must approximate 0. This enforces that x_2 has been used as much as stated by the grade r.

Products The right rule for products $\otimes_R^-$ behaves similarly to the $\multimap_L^-$ rule, passing the entire input context Γ to the first premise. This is in then used to synthesise the first sub-term of the pair t_1, yielding an output context Δ_1, which is passed to the second premise. After synthesising the second sub-term t_2, the output context for this premise becomes the output context of the rule's conclusion.

The left rule equivalent $\otimes_L^-$ binds two assumptions $x_1 : A$ $x_2 : B$ in the premise, representing the constituent sides of the pair. As with $\multimap_L^-$, we also ensure that these bound assumptions must not present in the premise's output context Δ.

$$\frac{\Gamma \vdash A \Rightarrow^- t_1 \mid \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow^- t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^- (t_1, t_2) \mid \Delta_2}\ \otimes_R^-$$

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^- t_2 \mid \Delta \qquad x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^- \textbf{let}\,(x_1, x_2) = x_3\,\textbf{in}\,t_2 \mid \Delta}\ \otimes_L^-$$

Sums The $\oplus_L^-$ rule synthesises the left and right branches of a case statement that may use resources differently. The output context therefore takes the *greatest lower bound* ($\sqcap$) of Δ_1 and Δ_2, given by definition 3.3.2,

Definition 3.3.2 (Partial greatest-lower bounds of contexts)**.** For all Γ_1, Γ_2:

$$\Gamma_1 \sqcap \Gamma_2 = \begin{cases} \emptyset & \Gamma_1 = \emptyset & \wedge\ \Gamma_2 = \emptyset \\ (\emptyset \sqcap \Gamma_2'), x :_{0 \sqcap s} A & \Gamma_1 = \emptyset & \wedge\ \Gamma_2 = \Gamma_2', x :_s A \\ (\Gamma_1' \sqcap (\Gamma_2', \Gamma_2'')), x : A & \Gamma_1 = \Gamma_1', x : A & \wedge\ \Gamma_2 = \Gamma_2', x : A, \Gamma_2'' \\ (\Gamma_1' \sqcap (\Gamma_2', \Gamma_2'')), x :_{r \sqcap s} A & \Gamma_1 = \Gamma_1', x :_r A & \wedge\ \Gamma_2 = \Gamma_2', x :_s A, \Gamma_2'' \end{cases}$$

where $r \sqcap s$ is the greatest-lower bound of grades r and s if it exists, derived from $\sqsubseteq$.

$$\frac{\Gamma \vdash A \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{-} \mathbf{inl}\, t \mid \Delta}\oplus 1_{R}^{-} \quad \frac{\Gamma \vdash B \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{-} \mathbf{inr}\, t \mid \Delta}\oplus 2_{R}^{-}$$

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \quad \Gamma, x_3 : B \vdash C \Rightarrow^{-} t_2 \mid \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2|}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^{-} \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \rightarrow t_1;\ \mathbf{inr}\ x_3 \rightarrow t_2 \mid \Delta_1 \sqcap \Delta_2}\oplus_{L}^{-}$$

As an example of $\sqcap$, consider the semiring of intervals over natural numbers and two judgements that could be used as premises for the $(\oplus_{L}^{-})$ rule:

$$\Gamma, y :_{[0...5]} A', x_2 : A \vdash C \Rightarrow^{-} t_1 \mid y :_{[2...5]} A'$$
$$\Gamma, y :_{[0...5]} A', x_3 : B \vdash C \Rightarrow^{-} t_2 \mid y :_{[3...4]} A'$$

where t_1 uses y such that there are 2-5 uses remaining and t_2 uses y such that there are 3-4 uses left. To synthesise **case** x_1 **of inl** $x_2 \rightarrow t_1$; **inr** $x_3 \rightarrow t_2$ the output context must be pessimistic about what resources are left, thus we take the greatest-lower bound yielding the interval $[2\ldots4]$ here: we know y can be used at least twice and at most 4 times in the rest of the synthesised program.

Unit The right and left rules for units are then self-explanatory following the subtractive resource model:

$$\frac{}{\Gamma \vdash 1 \Rightarrow^{-} () \mid \Gamma}1_{R}^{-} \quad \frac{\Gamma \vdash C \Rightarrow^{-} t \mid \Delta}{\Gamma, x : 1 \vdash C \Rightarrow^{-} \mathbf{let}\, () = x\, \mathbf{in}\, t \mid \Delta}1_{L}^{-}$$

This completes subtractive synthesis. We conclude with a key result, that synthesised terms are well-typed at the type from which they were synthesised:

Lemma 3.3.1 (Subtractive synthesis soundness). *For all* Γ *and* A *then:*

$$\Gamma \vdash A \Rightarrow^{-} t \mid \Delta \quad \implies \quad \Gamma - \Delta \vdash t : A$$

i.e. t *has type* A *under context* $\Gamma - \Delta$*, that contains just those linear and graded variables with grades reflecting their use in* t*. Appendix B.2 provides the proof.*

3.3.2 Additive Resource Management

We now present the dual to subtractive resource management — the *additive* approach. Additive synthesis also uses the input-output context approach, but where output contexts describe exactly which assumptions were used to synthesise a term, rather than which assumptions are still available. Additive synthesis rules are read bottom-up, with $\Gamma \vdash A \Rightarrow^{+} t; \Delta$ meaning that from the type A we synthesise a term t using exactly the assumptions Δ that originate from the input context Γ.

Variables We unpack the rules, starting with variables:

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^{+} x;\ x : A}\ \textsc{LinVar}^{+} \qquad \frac{}{\Gamma, x :_r A \vdash A \Rightarrow^{+} x;\ x :_1 A}\ \textsc{GrVar}^{+}$$

For a linear assumption, the output context contains just the variable that was synthesised. For a graded assumption $x :_r A$, the output context contains the assumption graded by 1. To synthesise a variable from a graded assumption, we must check that the use is compatible with the grade.

Graded modalities The subtractive approach handled the $\textsc{GrVar}^{-}$ by a constraint $\exists s.\, r \sqsupseteq s + 1$. Here however, the point at which we check that a graded assumption has been used according to the grade takes place in the $\Box_L^{+}$rule, where graded assumptions are bound:

$$\frac{\Gamma, x_2 :_r A \vdash B \Rightarrow^+ t;\, \Delta \qquad \textit{if } x_2 :_s A \in \Delta \textit{ then } s \sqsubseteq r \textit{ else } 0 \sqsubseteq r}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^+ \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t;\, (\Delta \backslash x_2), x_1 : \Box_r A}\;\Box_L^+$$

Here, t is synthesised under a fresh graded assumption $x_2 :_r A$. This produces an output context containing x_2 with some grade s that describes how x_2 is used in t. An additional premise requires that the original grade r approximates either s if x_2 appears in Δ or 0 if it does not, ensuring that x_2 has been used correctly. For the $\mathbb{N}$-semiring with equality as the ordering, this would ensure that a variable has been used exactly the number of times specified by the grade.

The synthesis of a promotion is considerably simpler in the additive approach. In subtractive resource management it was necessary to calculate how resources were used in the synthesis of t before then applying the scalar context multiplication by the grade r and subtracting this from the original input Γ. In additive resource management, however, we can simply apply the multiplication directly to the output context Δ to obtain how our assumptions are used in $[t]$:

$$\frac{\Gamma \vdash A \Rightarrow^+ t;\, \Delta}{\Gamma \vdash \Box_r A \Rightarrow^+ [t];\, r \cdot \Delta}\;\Box_R^+$$

Functions Right and left rules for $\multimap$ have a similar shape to the subtractive calculus:

$$\frac{\Gamma, x : A \vdash B \Rightarrow^+ t;\, \Delta, x : A}{\Gamma \vdash A \multimap B \Rightarrow^+ \lambda x.t;\, \Delta}\;\multimap_R^+$$

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1;\, \Delta_1, x_2 : B \qquad \Gamma \vdash A \Rightarrow^+ t_2;\, \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^+ [(x_1\, t_2)/x_2]t_1;\, (\Delta_1 + \Delta_2), x_1 : A \multimap B}\;\multimap_L^+$$

Synthesising an abstraction ($\multimap^+_R$) requires that $x : A$ is in the output context of the premise, ensuring that linearity is preserved. Likewise for application ($\multimap^+_L$), the output context of the first premise must contain the linearly bound $x_2 : B$ and the final output context must contain the assumption being used in the application $x_1 : A \multimap B$. This output context computes the *context addition* (Def. 5.1.2) of both output contexts of the premises $\Delta_1 + \Delta_2$. If Δ_1 describes how assumptions were used in t_1 and Δ_2 respectively for t_2, then the addition of these two contexts describes the usage of assumptions for the entire subprogram. Recall, context addition ensures that a linear assumption may not appear in both Δ_1 and Δ_2, preventing us from synthesising terms that violate linearity.

Dereliction As in the subtractive calculus, we avoid duplicating left rules to match graded assumptions by giving a synthesising version of dereliction:

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^+ t;\ \Delta, y : A}{\Gamma, x :_s A \vdash B \Rightarrow^+ [x/y]t;\ \Delta + x :_1 A}\ \textsc{der}^+$$

The fresh linear assumption $y : A$ must appear in the output context of the premise, ensuring it is used. The final context therefore adds to Δ an assumption of x graded by 1, accounting for this use of x (temporarily renamed to y).

Products The right rule for products $\otimes^+_R$ follows the same structure as its subtractive equivalent, however, here Γ is passed to both premises. The conclusion's output context is then formed by taking the context addition of the Δ_1 and Δ_2.

The left rule, $\otimes^+_L$ follows fairly straightforwardly from the resource scheme.

$$\frac{\Gamma \vdash A \Rightarrow^+ t_1;\, \Delta_1 \qquad \Gamma \vdash B \Rightarrow^+ t_2;\, \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^+ (t_1, t_2);\, \Delta_1 + \Delta_2}\ \otimes^+_R$$

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^+ t_2;\, \Delta, x_1 : A, x_2 : B}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^+ \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2;\, \Delta, x_3 : A \otimes B}\ \otimes^+_L$$

Sums In contrast to the subtractive rule, the rule $\oplus^+_L$ takes the least-upper bound of the premise's output contexts (see definition 3.1.1). Otherwise, the right and left rules for synthesising programs from sum types are straightforward.

$$\frac{\Gamma \vdash A \Rightarrow^+ t;\, \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inl}\,t;\, \Delta}\ \oplus 1^+_R \qquad \frac{\Gamma \vdash B \Rightarrow^+ t;\, \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inr}\,t;\, \Delta}\ \oplus 2^+_R$$

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow^+ t_1;\, \Delta_1, x_2 : A \qquad \Gamma, x_3 : B \vdash C \Rightarrow^+ t_2;\, \Delta_2, x_3 : B}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^- \mathbf{case}\ x_1\ \mathbf{of}\ \mathbf{inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcup \Delta_2, x_1 : A \oplus B}\ \oplus^+_L$$

Unit As in the subtractive approach, the right and left rules for unit types, are as expected.

$$\frac{}{\Gamma \vdash \mathbf{1} \Rightarrow^+ ();\, \emptyset}\ 1^+_R \qquad \frac{\Gamma \vdash C \Rightarrow^+ t;\, \Delta}{\Gamma, x : \mathbf{1} \vdash C \Rightarrow^+ \mathbf{let}\,() = x\,\mathbf{in}\,t;\, \Delta, x : \mathbf{1}}\ 1^+_L$$

Thus concludes the rules for additive synthesis. As with subtractive, we have prove that this calculus is sound.

Lemma 3.3.2 (Additive synthesis soundness)**.** *For all* Γ *and* A*:*

$$\Gamma \vdash A \Rightarrow^+ t;\, \Delta \quad \implies \quad \Delta \vdash t : A$$

Appendix **??** *gives the proof.*

Thus, the synthesised term t is well-typed at A using only the assumptions Δ. , where Δ is a subset of Γ. i.e., synthesised terms are well typed at the type from which they were synthesised.

Additive pruning

As seen above, the additive approach delays checking whether a variable is used according to its linearity/grade until it is bound. We hypothesise that this can lead additive synthesis to explore many ultimately ill-typed (or *ill-resourced*) paths for too long. Subsequently, we define a "pruning" variant of any additive rules with multiple sequenced premises. For $\otimes_R^+$ this is:

$$\frac{\Gamma \vdash A \Rightarrow^+ t_1;\, \Delta_1 \qquad \Gamma - \Delta_1 \vdash B \Rightarrow^+ t_2;\, \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^+ (t_1, t_2);\, \Delta_1 + \Delta_2}\ \otimes_R'^{+}$$

Instead of passing Γ to both premises, Γ is the input only for the first premise. This premise outputs context Δ_1 that is subtracted from Γ to give the input context of the second premise. This provides an opportunity to terminate the current branch of synthesis early if $\Gamma - \Delta_1$ does not contain the necessary resources to attempt the second premise. The $\multimap_L^+$ rule is similarly adjusted:

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1;\, \Delta_1, x_2 : B \qquad \Gamma - \Delta_1 \vdash A \Rightarrow^+ t_2;\, \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^+ [(x_1\, t_2)/x_2]t_1;\, (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap_L'^{+}$$

Lemma 3.3.3 (Additive pruning synthesis soundness)**.** *For all* Γ *and* A*:*

$$\Gamma \vdash A \Rightarrow^+ t;\, \Delta \quad \Longrightarrow \quad \Delta \vdash t : A$$

Appendix **??** *gives the proof.*

3.4 Focusing

3.5 Evaluation

Prior to evaluation, we made the following hypotheses about the relative performance of the additive versus subtractive approaches:

1. Additive synthesis should make fewer calls to the solver, with lower complexity theorems (fewer quantifiers). Dually, subtractive synthesis makes more calls to the solver with higher complexity theorems (more quantifiers);
2. For complex problems, additive synthesis will explore more paths as it cannot tell whether a variable is not well-resourced until closing a binder; additive pruning and subtractive will explore fewer paths as they can fail sooner.
3. A corollary of the above two: simple examples will likely be faster in additive mode, but more complex examples will be faster in subtractive mode.

Methodology

We implemented our approach as a synthesis tool for Granule, integrated with its core tool. Granule features ML-style polymorphism (rank-0 quantification) but we do not address polymorphism here. Instead, programs are synthesised from type schemes treating universal type variables as logical atoms.

Constraints on resource usage are handled via Granule's existing symbolic engine, which compiles constraints on grades (for various semirings) to the SMT-lib format for Z3 **?**. We use the LogicT monad for backtracking search **?** and the Scrap Your Reprinter library for splicing synthesised code into syntactic "holes", preserving the rest of the program text **?**.

To evaluate our synthesis tool we developed a suite of benchmarks comprising Granule type schemes for a variety of operations using linear and graded modal types. We divide our benchmarks into several classes of problem:

- **Hilbert**: the Hilbert-style axioms of intuitionistic logic (including SKI combinators), with appropriate $\mathbb{N}$ and $\mathbb{N}$-intervals grades where needed (see, e.g., S combinator in Example 2.1.1 or coproduct elimination in Example 3.1.1).
- **Comp**: various translations of function composition into linear logic: multiplicative, call-by-value and call-by-name using ! ?, I/O using ! ?, and coKleisli composition over $\mathbb{N}$ and arbitrary semirings: e.g. $\forall r, s \in \mathcal{R}$:

$$\textit{comp-coK}_{\mathcal{R}} : \Box_r(\Box_s A \to B) \to (\Box_r B \to C) \to \Box_{r \cdot s} A \to C$$

- **Dist**: distributive laws of various graded modalities over functions, sums, and products ?, e.g., $\forall r \in \mathbb{N}$, or $\forall r \in \mathcal{R}$ in any semiring, or $r = [0...\infty]$:

$$\textit{pull}_{\oplus} : (\Box_r A \oplus \Box_r B) \to \Box_r(A \oplus B) \qquad \textit{push}_{\multimap} : \Box_r(A \to B) \to \Box_r A \to \Box_r B$$

- **Vec**: map operations on vectors of fixed size encoded as products, e.g.:

$$\textit{vmap}_5 : \Box_5(A \to B) \to ((((A \otimes A) \otimes A) \otimes A) \otimes A) \to ((((B \otimes B) \otimes B) \otimes B) \otimes B)$$

- **Misc**: includes Example 3.1.2 (information-flow security) and functions which must share or split resources between graded modalities, e.g.:

$$\textit{share} : \Box_4 A \to \Box_6 A \to \Box_2(((((A \otimes A) \otimes A) \otimes A) \otimes A) \to B) \to (B \otimes B)$$

Appendix B.1 lists the type schemes for these synthesis problems (32 in total).

We found that Z3 is highly variable in its solving time, so timing measurements are computed as the mean of 20 trials. We used Z3 version 4.8.8 on a Linux laptop with an Intel i7-8665u @ 4.8 Ghz and 16 Gb of RAM.

Results and analysis

For each synthesis problem, we recorded whether synthesis was successful or not (denoted ✓ or ×), the mean total synthesis time (μT), the mean total time spent by the SMT solver (μSMT), and the number of calls made to the SMT solver (N). Table 2 summarises the results with the fastest case for each benchmark highlighted. For all benchmarks that used the SMT solver, the solver accounted for $91.73\% - 99.98\%$ of synthesis time, so we report only the mean total synthesis time μT. We set a timeout of 120 seconds.

Additive versus subtractive As expected, the additive approach generally synthesises programs faster than the subtractive. Our first hypothesis (that the additive approach in general makes fewer calls to the SMT solver) holds for almost all benchmarks, with the subtractive approach often far exceeding the number made by the additive. This is explained by the difference in graded variable synthesis between approaches. In the additive, a constant grade 1 is given for graded assumptions in the output context, whereas in the subtractive, a fresh grade variable is created with a constraint on its usage which is checked immediately. As the total synthesis time is almost entirely spent in the SMT solver (more than 90%), solving constraints is by far the most costly part of synthesis leading to the additive approach synthesising most examples in a shorter amount of time.

Graded variable synthesis in the subtractive case also results in several examples failing to synthesise. In some cases, e.g., the first three *comp* benchmarks, the subtractive approach times-out as synthesis diverges with constraints growing in size due to the maximality condition and absorbing behaviour of $[0...\infty]$ interval. In the case of *coK*-$\mathcal{R}$ and *coK*-$\mathbb{N}$, the generated constraints have the form $\forall r.\exists s.r \sqsupseteq s + 1$ which is not valid $\forall r \in \mathbb{N}$ (e.g., when $r = 0$), which suggests that the subtractive approach does not work well for polymorphic grades. As further work, we are considering an alternate rule for synthesising promotion with

		Additive			Additive (pruning)			Subtractive		
	Problem		μT (ms)	N		μT (ms)	N		μT (ms)	N
Hilbert	$\otimes$Intro	✓	6.69 (0.05)	2	✓	9.66 (0.23)	2	✓	10.93 (0.31)	2
	$\otimes$Elim	✓	0.22 (0.01)	0	✓	0.05 (0.00)	0	✓	0.06 (0.00)	0
	$\oplus$Intro	✓	0.08 (0.00)	0	✓	0.07 (0.00)	0	✓	0.07 (0.00)	0
	$\oplus$Elim	✓	7.26 (0.30)	2	✓	13.25 (0.58)	2	✓	204.50 (8.78)	15
	SKI	✓	8.12 (0.25)	2	✓	24.98 (1.19)	2	✓	41.92 (2.34)	4
Comp	01	✓	28.31 (3.09)	5	✓	41.86 (0.38)	5	×	Timeout	-
	cbn	✓	13.12 (0.84)	3	✓	26.24 (0.27)	3	×	Timeout	-
	cbv	✓	19.68 (0.98)	5	✓	34.15 (0.98)	5	×	Timeout	-
	$\circ coK_{\mathcal{R}}$	✓	33.37 (2.01)	2	✓	27.37 (0.78)	2	×	92.71 (2.37)	8
	$\circ coK_{\mathbb{N}}$	✓	27.59 (0.67)	2	✓	21.62 (0.59)	2	×	95.94 (2.21)	8
	mult	✓	0.29 (0.02)	0	✓	0.12 (0.00)	0	✓	0.11 (0.00)	0
Dist	$\otimes$-!	✓	12.96 (0.48)	2	✓	32.28 (1.32)	2	✓	10487.92 (4.38)	7
	$\otimes$-$\mathbb{N}$	✓	24.83 (1.01)	2	×	32.18 (0.80)	2	×	31.33 (0.65)	2
	$\otimes$-$\mathcal{R}$	✓	28.17 (1.01)	2	×	29.72 (0.90)	2	×	31.91 (1.02)	2
	$\oplus$-!	✓	7.87 (0.23)	2	✓	16.54 (0.43)	2	✓	160.65 (2.26)	4
	$\oplus$-$\mathbb{N}$	✓	22.13 (0.70)	2	✓	30.30 (1.02)	2	×	23.82 (1.13)	1
	$\oplus$-$\mathcal{R}$	✓	22.18 (0.60)	2	✓	31.24 (1.40)	2	×	16.34 (0.40)	1
	$\multimap$-!	✓	6.53 (0.16)	2	✓	10.01 (0.25)	2	✓	342.52 (2.64)	4
	$\multimap$-$\mathbb{N}$	✓	29.16 (0.82)	2	✓	28.71 (0.67)	2	×	54.00 (1.53)	4
	$\multimap$-$\mathcal{R}$	✓	29.31 (1.84)	2	✓	27.44 (0.60)	2	×	61.33 (2.28)	4
Vec	vec5	✓	4.72 (0.07)	1	✓	14.93 (0.21)	1	✓	78.90 (2.25)	6
	vec10	✓	5.51 (0.36)	1	✓	20.81 (0.77)	1	✓	142.87 (5.86)	11
	vec15	✓	9.75 (0.25)	1	✓	22.09 (0.24)	1	✓	195.24 (3.20)	16
	vec20	✓	13.40 (0.46)	1	✓	30.18 (0.20)	1	✓	269.52 (4.25)	21
Misc	split$\oplus$	✓	3.79 (0.04)	1	✓	5.10 (0.16)	1	✓	10732.65 (8.01)	6
	split$\otimes$	✓	14.07 (1.01)	3	✓	46.27 (2.04)	3	×	Timeout	-
	share	✓	292.02 (11.37)	44	✓	100.85 (2.44)	6	✓	193.33 (4.46)	17
	exm. 3.1.2	✓	8.09 (0.46)	2	✓	26.03 (1.21)	2	✓	284.76 (0.31)	3

Table 2: Results. μT in *ms* to 2 d.p. with standard sample error in brackets

constraints of the form $\exists s.s = s' * r$, i.e., a multiplicative inverse constraint.

In more complex examples we see evidence to support our second hypothesis. The *share* problem requires a lot of graded variable synthesis which is problematic for the additive approach, for the reasons described in the second hypothesis. In contrast, the subtractive approach performs better, with $\mu T = 193.3ms$ as opposed to additive's $292.02ms$. However, additive pruning outperforms both.

Additive pruning The pruning variant of additive synthesis (where subtraction takes place in the premises of multiplicative rules) had mixed results compared to the default. In simpler examples, the overhead of pruning (requiring SMT solving) outweighs the benefits obtained from reducing the space. However, in more complex examples which involve synthesising many graded variables (e.g. *share*), pruning is especially powerful, performing better than the subtractive approach. However, additive pruning failed to synthesis two examples which are polymorphic in their grade ($\otimes$-$\mathbb{N}$) and in the semiring/graded-modality ($\otimes$-$\mathcal{R}$).

Overall, the additive approach outperforms the subtractive and is successful at synthesising more examples, including ones polymorphic in grades and even the semiring itself. Given that the literature on linear logic theorem proving is typically subtractive, this is an interesting result. Going forward, we will focus on the additive scheme.

3.6 Conclusion

We have now laid out the foundations for a full program synthesis tool for Granule, exploring in particular the challenges presented by the treatment of data as resources with usage constraints, and how the synhtesis tool must manage these resources. Two schemes were proposed as a solution to this problem: additive and subtractive, with additive generally outperforming subtractive.

Going forward, we focus on the additive resource management scheme as the basis for synthesis with resources.

While the tool presented in this chapter allows users to synthesise a considerable subset of Granule programs, it is still quite limited in its expressivity. Data types comprise only product, sum, and unit types, while synthesis of recursive function defintions or functions which make use of other in-scope values such as top-level definitions are not permitted.

The features desrcibed above would all make theoretically feasible additions to the current calculi. However, there are limitations for which a solution would require significant alterations to the structure of the calculi. For example, synthesis of programs which perform nested pattern matching over graded values

In chapter ... a full synthesis tool for Granule with all these features is described.

However, the most notable limitation of our tool is the inability to synthesise programs which distribute a grade over a data type. Consider the archetypal distributive program *push*:

$$\mathit{push} : \Box_r(A \otimes B) \multimap \Box_r A \otimes \Box_r B$$

which takes a data type graded by r (in this case the product type $A \otimes B$), and *distributes* r over the constituent elements of the product A, and B. Given this goal type, how would we go about synthesising a program in our tool?

To demonstrate the issue, we instatiate the $\multimap_R^+$ rule at this type, building a partial synthesis tree. Note that although we use the additive scheme for this example, the same situation arises in the subtractive.

$$\dfrac{\dfrac{x_2 :_r A \otimes B \vdash \Box_r A \otimes \Box_r B \Rightarrow\ ?\ |\ ?}{x_1 : \Box_r(A \otimes B) \vdash \Box_r A \otimes \Box_r B \Rightarrow \textbf{ let } [x_2] = x_1 \textbf{ in } ?\ |\ ?}\,\Box_L^+}{\emptyset \vdash \Box_r(A \otimes B) \multimap \Box_r A \otimes \Box_r B \Rightarrow \lambda x_1.?\ |\ ?}\,\multimap_R^+$$

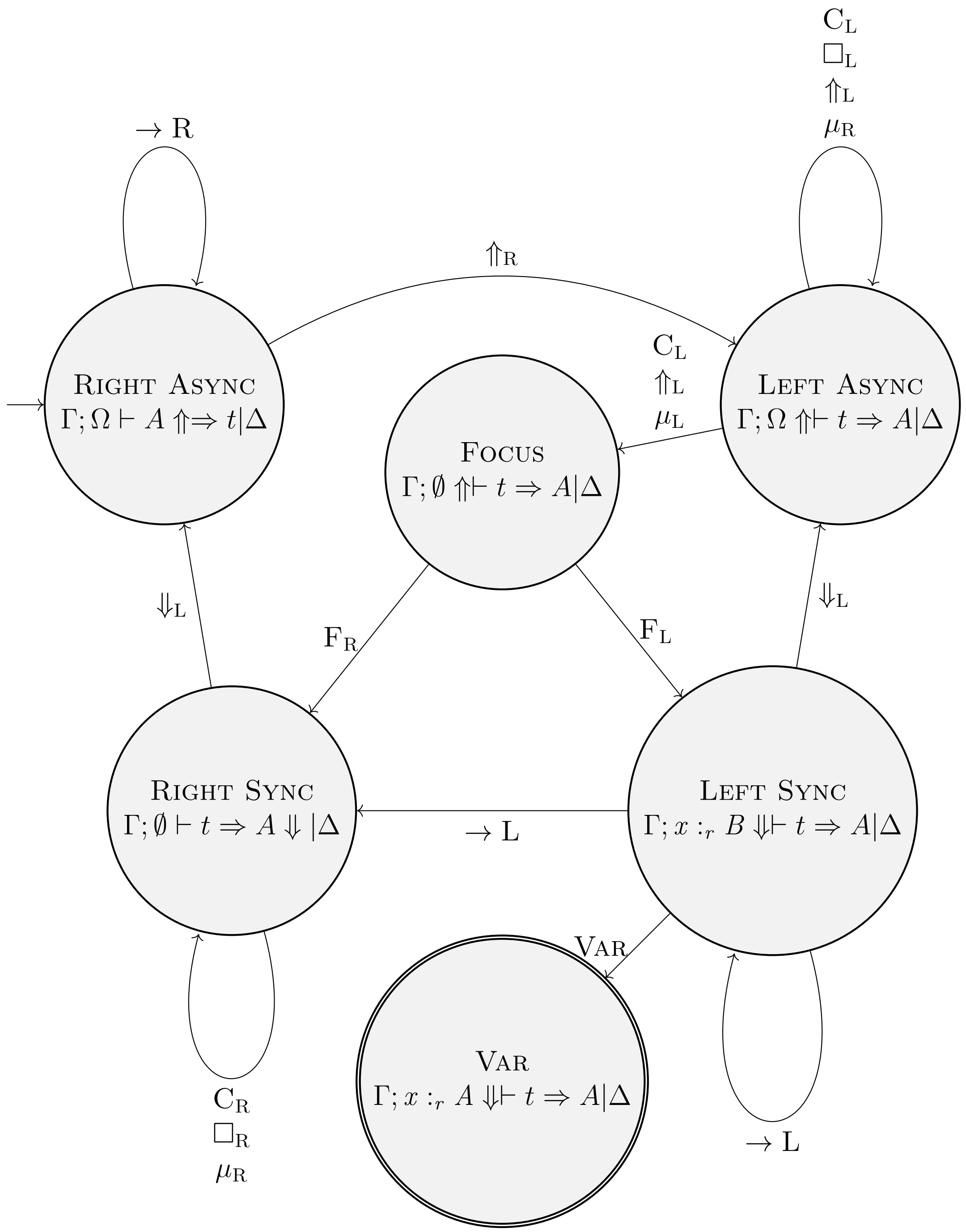
After applying $\multimap_R^+$ followed by $\Box_L^+$, we now have the graded assumption $x_2 :_r A \otimes B$ in our context which we must use to construct a term of type $\Box_r A \otimes \Box_r B$. We might expect that the path synthesis should take now would be to break x_2 down into two graded assumptions with types A and B, promote these graded assumptions using the $\Box_R^+$ rule, before finally peforming a pair introduction to yield $\Box_r A \otimes \Box_r B$. However, in order to apply the pair elimination rule $\otimes_L^+$ and break our graded assumption into two, we must perform a dereliction on x_2, to yield a linear copy:

$$\frac{x_2 :_r A \otimes B, x_3 : A \otimes B \vdash \Box_r A \otimes \Box_r B \Rightarrow \,?}{x_2 :_r A \otimes B \vdash \Box_r A \otimes \Box_r B \Rightarrow \,? \mid \,?}\ \textsc{der}^{+}$$

Clearly, this is not going to lead us to the goal: the $\Box_R^+$ rule cannot promote terms using linear assumptions. Therefore, *push* and other types which exhibit this distributive behaviour are not synthesiseable in our calculus.

One possible solution to this is to extend our calculi with additional rules

In the following chapter, we will present an alternative approach to generating programs which exhibit this distributive behaviour using a generic programming methodology. While this technique is much less expressive in general than our synthesis calculi here, it does address some of the areas in which our tool struggles. In particular, both synthesis and the deriving mechanism may be used in conjunction.



$\to$ R
C$_{\mathrm{L}}$ $\Box_{\mathrm{L}}$ $\Uparrow_{\mathrm{L}}$ μ_{R}
$\Uparrow_{\mathrm{R}}$
RIGHT ASYNC $\Gamma; \Omega \vdash A \Uparrow\Rightarrow t|\Delta$
C$_{\mathrm{L}}$ $\Uparrow_{\mathrm{L}}$ μ_{L}
LEFT ASYNC $\Gamma; \Omega \Uparrow\vdash t \Rightarrow A|\Delta$
FOCUS $\Gamma; \emptyset \Uparrow\vdash t \Rightarrow A|\Delta$
$\Downarrow_{\mathrm{L}}$
$\Downarrow_{\mathrm{L}}$
F$_{\mathrm{R}}$
F$_{\mathrm{L}}$
RIGHT SYNC $\Gamma; \emptyset \vdash t \Rightarrow A \Downarrow |\Delta$
$\to$ L
LEFT SYNC $\Gamma; x :_r B \Downarrow\vdash t \Rightarrow A|\Delta$
VAR
VAR $\Gamma; x :_r A \Downarrow\vdash t \Rightarrow A|\Delta$
C$_{\mathrm{R}}$ $\Box_{\mathrm{R}}$ μ_{R}
$\to$ L

Figure 5: Focusing State Machine

Chapter 4

Deriving graded combinators

When programming with graded modal types, we have observed there is often a need to 'distribute' a graded modality over a type, and vice versa, in order to compose programs. That is, we may find ourselves in possession of a $\Box_r(\mathsf{F}\alpha)$ value (for some parametric data type F) which needs to be passed to a pre-existing function (of our own codebase or a library) which requires a $\mathsf{F}(\Box_r\alpha)$ value, or perhaps vice versa. A *distributive law* (in the categorical sense, e.g., **?**) provides a conversion from one to the other. In this paper, we present a procedure to automatically synthesise these distributive operators, applying a generic programming methodology **?** to compute these operations given the base type (e.g., $\mathsf{F}\alpha$ in the above description). This serves to ease the use of graded modal types in practice, removing boilerplate code by automatically generating these 'interfacing functions' on-demand, for user-defined data types as well as built-in types.

Whilst this work can be viewed as synthesis, it is

Throughout, we refer to distributive laws of the form $\Box_r(\mathsf{F}\alpha) \rightarrow \mathsf{F}(\Box_r\alpha)$ as *push* operations (as they 'push' the graded modality inside the type constructor F), and dually $\mathsf{F}(\Box_r\alpha) \rightarrow \Box_r(\mathsf{F}\alpha)$ as *pull* operations (as they 'pull' the graded modality outside the type constructor F).

The primary contributions of this paper are then:

- an overview of the application of distributive laws in the context of graded modal types;
- an automatic procedure for calculating distributive laws from types and a formal analysis of their properties;
- a realisation of this approach in both Granule (embedded into the compiler) and Linear Haskell (expressed within the language itself, leveraging Haskell's advanced features);
- and derivations of related combinators for structural use of values in a graded linear setting.

Along the way, we also analyse choices made around the typed-analysis of pattern matching in various recent works on graded modal types. Through the comparison between Granule and Linear Haskell, we also highlight ways in which Linear Haskell could be made more general and flexible in the future.

Section **??** defines a core calculus GrMini^P with linear and graded modal types which provides an idealised, simply-typed subset of Granule with which we develop the core contribution. Section 4.2 gives the procedures for deriving *push* and *pull* operators for the core calculus, and verifies that these are distributive laws of endofunctors over the $\Box_r$ graded comonadic modality. Section **??** describes the details of how these procedures are realised in the Granule language. Section **??** relates this work to Linear Haskell, and demonstrates how the *push* and *pull* combinators for user-defined data types may be automatically generated at compile-time using Template Haskell. Section 4.5 gives a comparison of the recent literature with regards the typed (graded) analysis of pattern matching, which is germane to the typing and derivation of our distributive laws. Section 4.6 covers how other structural combinators for Granule and Linear Haskell may be derived. Finally, Section **??** discusses more related and future work.

We start with an extended motivating example typifying the kind of software engineering impedance problem that distributive laws solve. We use Granule since it is the main vehicle for the developments here, and introduce some of the key concepts of graded modal types (in a linear context) along the way.

4.0.1 Motivating Example

Consider the situation of projecting the first element of a pair. In Granule, this first-projection is defined and typed as the following polymorphic function (whose syntax is reminiscent of Haskell or ML):

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

Linearity is the default, so this represents a linear function applied to linear values.[1] However, the second component of the pair has a *graded modal type*, written `b [0]`, which means that we can use the value "inside" the graded modality 0 times by first 'unboxing' this capability via the pattern match `[y]` which allows weakening to be applied in the body to discard `y` of type `b`. In calculus of Section **??**, we denote '`b [0]`' as the type $\Box_0 b$ (Granule's graded modalities are written postfix with the 'grade' inside the box).

The type for `fst` is however somewhat restrictive: what if we are trying to use such a function with a value (call it `myPair`) whose type is not of the form `(a, b [0])` but rather `(a, b) [r]` for some grading term `r` which permits weakening? Such a situation readily arises when we are composing functional code, say between libraries or between a library and user code. In this situation, `fst myPair` is ill-typed. Instead, we could define a different first projection function for use with `myPair : (a, b) [r]` as:

```
fst' : ∀ {a b : Type, s : Semiring, r : s} . {0 ⩽ r} ⇒ (a, b) [r] → a
fst' [(x, y)] = x
```

[1]Granule uses → syntax rather than ⊸ for linear functions for the sake of familiarity with standard functional languages

This implementation uses various language features of Granule to make it as general as possible. Firstly, the function is polymorphic in the grade `r` and in the semiring `s` of which `r` is an element. Next, a refinement constraint `0 ⩽ r` specifies that by the preordering ⩽ associated with the semiring `s`, that `0` is approximated by `r` (essentially, that `r` permits weakening). The rest of the type and implementation looks more familiar for computing a first projection, but now the graded modality is over the entire pair.

From a software engineering perspective, it is cumbersome to create alternate versions of generic combinators every time we are in a slightly different situation with regards the position of a graded modality. Fortunately, this is an example to which a general *distributive law* can be deployed. In this case, we could define the following distributive law of graded modalities over products, call it `pushPair`:

```
pushPair : ∀ {a b : Type, s : Semiring, r : s} . (a, b) [r] → (a [r], b [r])
pushPair [(x, y)] = ([x], [y])
```

This 'pushes' the graded modality `r` into the pair (via pattern matching on the modality and the pair inside it, and then reintroducing the modality on the right hand side via `[x]` and `[y]`), distributing the graded modality to each component. Given this combinator, we can now apply `fst (pushPair myPair)` to yield a value of type `a [r]`, on which we can then apply the Granule standard library function `extract`, defined:

```
  extract : ∀ {a : Type, s : Semiring, r : s} . {(1 : s) ⩽ r} ⇒ a [r] → a
  extract [x] = x
```

to get the original `a` value we desired:

```
extract (fst (pushPair myPair)) : a
```

The `pushPair` function could be provided by the standard library, and thus we have not had to write any specialised combinators ourselves: we have applied supplied combinators to solve the problem.

Now imagine we have introduced some custom data type `List` on which we have a *map* function:

```
data List a = Cons a (List a) | Nil

map : ∀ {a b : Type} . (a → b) [0..∞] → List a → List b
map [f] Nil = Nil;
map [f] (Cons x xs) = Cons (f x) (map [f] xs)
```

Note that, via a graded modality, the type of `map` specifies that the parameter function, of type `a → b` is non-linear, used between 0 and ∞ times. Imagine now we have a value `myPairList : (List (a, b)) [r]` and we want to map first projection over it. But `fst` expects `(a, b [0])` and even with `pushPair` we require `(a, b) [r]`. *We need another distributive law*, this time of the graded modality over the `List` data type. Since `List` was user-defined, we now have to roll our own `pushList` operation, and so we are back to having to make specialised combinators for our data types.

The crux of this paper is that such distributive laws can be automatically calculated given the definition of a type. With our Granule implementation of this approach (Section **??**), we can then solve this combination problem via the following composition of combinators:

```
map (extract . fst . push @(,)) (push @List myPairList) : List a
```

where the `push` operations are written with their base type via `@` (a type application) and whose definitions and types are automatically generated during type checking. Thus the `push` operation is a *data-type generic function* **?**. This generic function is defined inductively over the structure of types, thus a programmer can introduce a new user-defined algebraic data type and have the implementation of the generic distributive law derived automatically. This reduces both the initial and future effort (e.g., if an ADT definition changes or new ADTs are introduced).

Dual to the above, there are situations where a programmer may wish to *pull* a graded modality out of a structure. This is possible with a dual distributive law, which could be written by hand as:

```
pullPair : ∀ {a b : Type, s : Semiring, m n : s} . (a [n], b [m]) → (a, b) [n ⊓ 
pullPair ([x], [y]) = [(x, y)]
```

Note that the resulting grade is defined by the greatest-lower bound (meet) of `n` and `m`, if it exists as defined by a preorder for semiring `s` (that is, ⊓ is not a total operation). This allows some flexibility in the use of the *pull* operation when grades differ in different components but have a greatest-lower bound which can be 'pulled out'. Our approach also allows such operations to be generically derived.

4.1 Extending the Linear-base Calculus

$$t ::= x \mid t_1\, t_2 \mid \lambda x.t \mid [t] \mid C\, t_0 \ldots t_n \mid \mathbf{case}\ t\ \mathbf{of}\ p_1 \mapsto t_1; \ldots; p_n \mapsto t_n \mid \mathbf{letrec}\ x = t_1\ \mathbf{in}\ t_2 \qquad \text{(terms)}$$

$$p ::= x \mid \_ \mid [p] \mid C\, p_0 \ldots p_n \qquad \text{(patterns)}$$

$$A, B ::= A \multimap B \mid \alpha \mid A \otimes B \mid A \oplus B \mid \mathbf{1} \mid \Box_r A \mid \mu X.A \mid X \qquad \text{(types)}$$

$$C ::= () \mid \mathsf{inl} \mid \mathsf{inr} \mid (,) \qquad \text{(data constructors)}$$

4.2 Automatically Deriving *push* and *pull*

Now that we have established the core theory, we describe the algorithmic calculation of distributive laws in GrMini^P. Note that whilst GrMini^P is simply typed (monomorphic), it includes type variables (ranged over by α) to enable the distributive laws to be derived on parametric types. In the implementation, these will really be polymorphic type variables, but the derivation procedure need only treat them as some additional syntactic type construct.

Notation Let $\mathsf{F} : \mathsf{Type}^n \rightarrow \mathsf{Type}$ be an n-ary type constructor (i.e. a constructor which takes n type arguments), whose free type variables provide the n parameter types. We write $\mathsf{F}\overline{\alpha_i}$ for the application of F to type variables α_i for all $1 \leq i \leq n$.

Push We automatically calculate *push* for F applied to n type variables $\overline{\alpha_i}$ as the operation:

$$[\![\mathsf{F}\overline{\alpha_i}]\!]_{\mathsf{push}} : \Box_r \mathsf{F}\overline{\alpha_i} \multimap \mathsf{F}(\overline{\Box_r \alpha_i})$$

where we require $1 \sqsubseteq r$ *if* $|\mathsf{F}\overline{\alpha_i}| > 1$ due to the [PCON] rule (e.g., if F contains a sum).

For types A closed with respect to recursion variables, let $[\![A]\!]_{\mathsf{push}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{push}}\ z$

given by an intermediate interpretation $[\![A]\!]^{\Sigma}_{\mathsf{push}}$ where Σ is a context of *push* combinators for the recursive type variables:

$$[\![1]\!]^{\Sigma}_{\mathsf{push}}\ z = \texttt{<<no parses (char 14): case z of [u***nit .] -> unit . >>}$$

$$[\![\alpha]\!]^{\Sigma}_{\mathsf{push}}\ z = z$$

$$[\![X]\!]^{\Sigma}_{\mathsf{push}}\ z = \Sigma(X)\ z$$

$$[\![A \oplus B]\!]^{\Sigma}_{\mathsf{push}}\ z = \frac{\begin{array}{l} (C_i : {B_1}^{q^i_1} \to \dots \to {B_n}^{q^i_n} \to K\ \vec{A}) \in D \\ \Gamma, x :_r K\ \vec{A}, y^i_1 :_{r \cdot q^i_1} B_1, \dots, y^i_n :_{r \cdot q^i_1} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\ \vec{A}, y^i_1 :_{s^i_1} B_1, \dots, y^i_n \\ \exists {s'}^i_j .\ s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j \\ s_i = {s'}^i_1 \sqcup \dots \sqcup {s'}^i_n \\ |K\ \vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup \dots \sqcup s_m \end{array}}{\Gamma, x :_r K\ \vec{A} \vdash B \Rightarrow \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ y^i_1 \dots y^i_n \mapsto t_i} \mid (\Delta_1 \sqcup \dots \sqcup \Delta_m), x :_{(r_1 \sqcup \dots \sqcup r_m)+(s_1 \sqcup \dots}}$$

$$[\![A \otimes B]\!]^{\Sigma}_{\mathsf{push}}\ z = \frac{\begin{array}{l} (C_i : {B_1}^{q^i_1} \to \dots \to {B_n}^{q^i_n} \to K\ \vec{A}) \in D \\ \Gamma, x :_r K\ \vec{A}, y^i_1 :_{r \cdot q^i_1} B_1, \dots, y^i_n :_{r \cdot q^i_1} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\ \vec{A}, y^i_1 :_{s^i_1} B_1, \dots, y^i_n \\ \exists {s'}^i_j .\ s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j \\ s_i = {s'}^i_1 \sqcup \dots \sqcup {s'}^i_n \\ |K\ \vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup \dots \sqcup s_m \end{array}}{\Gamma, x :_r K\ \vec{A} \vdash B \Rightarrow \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ y^i_1 \dots y^i_n \mapsto t_i} \mid (\Delta_1 \sqcup \dots \sqcup \Delta_m), x :_{(r_1 \sqcup \dots \sqcup r_m)+(s_1 \sqcup \dots}}$$

$$[\![A \multimap B]\!]_{\mathsf{push}}\ z = \lambda y. \frac{\begin{array}{l} (C_i : {B_1}^{q^i_1} \to \dots \to {B_n}^{q^i_n} \to K\ \vec{A}) \in D \\ \Gamma, x :_r K\ \vec{A}, y^i_1 :_{r \cdot q^i_1} B_1, \dots, y^i_n :_{r \cdot q^i_1} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\ \vec{A}, y^i_1 :_{s^i_1} B_1, \dots, \\ \exists {s'}^i_j .\ s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j \\ s_i = {s'}^i_1 \sqcup \dots \sqcup {s'}^i_n \\ |K\ \vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup \dots \sqcup s_m \end{array}}{\Gamma, x :_r K\ \vec{A} \vdash B \Rightarrow \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ y^i_1 \dots y^i_n \mapsto t_i} \mid (\Delta_1 \sqcup \dots \sqcup \Delta_m), x :_{(r_1 \sqcup \dots \sqcup r_m)+(s}}$$

$$[\![\mu X.A]\!]^{\Sigma}_{\mathsf{push}}\ z = \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \texttt{<<no parses (char 54): \{mu X . \{[] r A\}\} -o \{\{(mu X . A)\} [[] r ai /}}_{\mathsf{push}}$$

In the case of *push* on a value of type $\mathbf{1}$, we pattern match on the value, eliminating the graded modality via the unboxing pattern match and returning the unit value.

For type variables, *push* is simply the identity of the value, while for recursion variables we lookup the X's binding in Σ and apply it to the value. For sum and product types, *push* works by pattern matching on the type's constructor(s) and then inductively applying *push* to the boxed arguments, re-applying them to the constructor(s). Unlike *pull* below, the *push* operation can be derived for function types, with a contravariant use of *pull*. For recursive types, we inductively apply *push* to the value with a fresh recursion variable bound in Σ, representing a recursive application of push.

There is no derivation of a distributive law for types which are themselves graded modalities (see further work discussion in Section **??**).

The appendix **?** proves that $[\![A]\!]_{\mathsf{push}}$ is type sound, i.e., its derivations are well-typed.

4.3 Implementation in Granule

The Granule type checker implements the algorithmic derivation of *push* and *pull* distributive laws as covered in the previous section. Whilst the syntax of GR-MINI^P types had unit, sum, and product types as primitives, in Granule these are provided by a more general notion of type constructor which can be extended by user-defined, generalized algebraic data types (GADTs). The procedure outlined in Section 4.2 is therefore generalised slightly so that it can be applied to any data type: the case for $A \oplus B$ is generalised to types with an arbitrary number of data constructors.

Our deriving mechanism is exposed to programmers via explicit (visible) type application (akin to that provided in GHC Haskell **?**) on reserved names `push` and `pull`. Written `push @T` or `pull @T`, this signals to the compiler that we wish to derive the corresponding distributive laws at the type `T`. For example, for the `List : Type → Type` data type from the standard library, we can write the expression `push @List` which the type checker recognises as a function of type:

```
push @List : ∀ {a : Type, s : Semiring, r : s} . {1 ⩽ r} ⇒ (List a) [r] → List
```

Note this function is not only polymorphic in the grade, but polymorphic in the semiring itself. Granule identifies different graded modalities by their semirings, and thus this operation is polymorphic in the graded modality. When the type checker encounters such a type application, it triggers the derivation procedure of Section 4.2, which also calculates the type. The result is then stored in the state of the frontend to be passed to the interpreter (or compiler) after type checking. The derived operations are memoized so that they need not be recalculated if a particular distributive law is required more than once. Otherwise, the implementation largely follows Section 4.2 without surprises, apart from some additional machinery for specialising the types of data constructors coming from (generalized) algebraic data types.

Examples Section **??** motivated the crux of this paper with a concrete example, which we can replay here in concrete Granule, using its type application technique for triggering the automatic derivation of the distributive laws. Previously, we defined `pushPair` by hand which can now be replaced with:

```
push @(,) : ∀ {a, b : Type, s : Semiring, r : s} . (a, b) [r] → (a [r], b [r])
```

Note that in Granule `(,)` is an infix type constructor for products as well as terms. We could then replace the previous definition of `fst’` from Section **??** with:

```
fst’ : ∀ {a, b : Type, r : Semiring} . {0 ⩽ r} ⇒ (a, b) [r] → a
fst’ = let [x’] = fst (push @(,) x) in x’
```

The point however in the example is that we need not even define this intermediate combinator, but can instead write the following wherever we need to compute the first projection of `myPair : (a, b) [r]`:

```
extract (fst (push @(,) myPair)) : a

```

We already saw that we can then generalise this by applying this first projection inside of the list
`myPairList : (List (a, b)) [r]` directly, using `push @List`.

In a slightly more elaborate example, we can use the `pull` combinator for pairs to implement a function that duplicates a pair (given that both elements can be consumed twice):

```
copyPair : ∀ {a, b : Type} . (a [0..2], b [2..4]) → ((a, b), (a, b))
copyPair x = copy (pull @(,) x) -- where, copy : a [2] → (a, a)

```

Note `pull` here computes the greated-lower bound of intervals `0..2` and `2..4` which is `2..2`, i.e., we can provide a pair of `a` and `b` values which can each be used exactly twice, which is what is required for `copy`.

As another example, interacting with Granule's indexed types (GADTs), consider a simple programming task of taking the head of a sized-list (vector) and duplicating it into a pair. The `head` operation is typed: `head : ∀ {a : Type, n : Nat} . (Vec (n +`
which has a graded modal input with grade `0..1` meaning the input vector is used 0 or 1 times: the head element is used once (linearly) for the return but the tail is discarded.

This head element can then be copied if it has this capability via a graded modality, e.g., a value of type `(Vec (n + 1) (a [2])) [0..1]` permits:

```
copyHead' : ∀ {a : Type, n : Nat} . (Vec (n + 1) (a [2])) [0..1] → (a, a)
copyHead' xs = let [y] = head xs in (y, y) -- [y] unboxes (a [2]) to y:a usable

```

Here we "unbox" the graded modal value of type `a [2]` to get a non-linear variable `y` which we can use precisely twice. However, what if we are in a programming context where we have a value `Vec (n + 1) a` with no graded modality on the type `a`? We can employ two idioms here: (i) take a value of type `(Vec (n + 1) a) [0..2]` and split its modality in two: `(Vec (n + 1) a) [2] [0..1]` (ii) then use *push* on the inner graded modality `[2]` to get `(Vec (n + 1) (a [2])) [0..1]`.

Using `push @Vec` we can thus write the following to duplicate the head element of a vector:

```
copyHead : ∀ {a : Type, n : Nat} . (Vec (n + 1) a) [0..2] → (a, a)
copyHead = copy . head . boxmap [push @Vec] . disject

```

which employs combinators from the standard library and the derived distributive law, of type:

```
boxmap    : ∀ {a b : Type, s : Semiring, r : s}        . (a  → b) [r] → a [r] -
disject   : ∀ {a : Type, s : Semiring, n m : s}        . a [m * n] → (a [n]) [m]
push @Vec : ∀ {a : Type, n : Nat, s : Semiring, r : s} . (Vec n a) [r] → Vec n (

```

4.4 Application to Linear Haskell

While Granule has been pushing the state-of-the-art in graded modal types, similar features have been added to more mainstream languages. Haskell has recently added support for linear types via an underlying graded system which enables linear types as a smooth extension to GHC's current type system **?**.[2] Functions may be linear with respect to their input (meaning, the function will consume its argument exactly once if the result is consumed exactly once), but they can also consume their argument `r` times for some multiplicity `r` via explicit 'multiplicity

[2]Released as part of GHC v9.0.1 in February 2021 `https://www.haskell.org/ghc/download_ghc_9_0_1.html`

polymorphism'. Unlike Granule, Linear Haskell limits this multiplicity to the set of either one or many (the paper speculates about extending this with zero)—full natural numbers or other semirings are not supported.

In Linear Haskell, the function type (`a %r -> b`) can be read as "a function from `a` to `b` which uses `a` according to `r`" where `r` is either 1 (also written as `'One`) or ω (written as `'Many`). For example, the following defines and types the linear and non-linear functions `swap` and `copy` in Linear Haskell:

```
{-# LANGUAGE LinearTypes #-}
import GHC.Types

swap :: (a %1 -> b %1 -> c) %1 -> (b %1 -> a %1 -> c)
swap f x y = f y x

copy :: a %'Many -> (a, a)
copy x = (x, x)
```

Assigning the type `a %1 -> (a, a)` to `copy` would result in a type error due to a mismatch in multiplicity.

The approach of Linear Haskell (as formally defined by Bernardy et al. **?**) resembles earlier coeffect systems **??** and more recent work on graded systems which have appeared since **??**. In these approaches there is no underlying linear type system (as there is in Granule) but grading is instead pervasive with function arrows carrying a grade describing the use of their parameter. Nevertheless, recent systems also provide a graded modality as this enables more fine-grained usage information to be ascribed to compound data. For example, without graded modalities it cannot be explained that the first projection on a pair uses the first component once and its second component not at all (instead a single grade would have to be assigned to the entire pair).

We can define a graded modality in Linear Haskell via the following `Box` data type that is parameterized over the multiplicity `r` and the value type `a`, defined as:

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

A `Box` type is necessary to make explicit the notion that a value may be consumed a certain number of times, where ordinarily Linear Haskell is concerned only with whether individual functions consume their arguments linearly or not. Thus, distributive laws of the form discussed in this paper then become useful in practice when working with Linear Haskell.

The `pushPair` and `pullPair` functions from Section **??** can then be implemented in Linear Haskell with the help of this `Box` type:

```
pushPair :: Box r (a, b) %1 -> (Box r a, Box r b)
pushPair (Box (x, y)) = (Box x, Box y)
```

```
pullPair :: (Box r a, Box r b) %1 -> Box r (a, b)
pullPair (Box x, Box y) = Box (x, y)
```

Interestingly, `pushPair` could also be implemented as a function of type `(a, b) %r -> (Box r a, Box r` and in general we can formulate push combinators typed `(f a) %r -> f (Box r a)`, i.e., consuming the input `r` times, returning a box with multiplicity `r`, but we stick with the above formulation for consistency.

While more sophisticated methods are outlined in this paper for automatically synthesizing these functions in the context of Granule, in the context of Linear Haskell (which has a simpler notion of grading) distributive laws over unary and binary type constructors can be captured with type classes:

```
class Pushable f where
  push :: Box r (f a) %1-> f (Box r a)
class Pushable2 f where
  push2 :: Box r (f a b) %1-> f (Box r a) (Box r b)

class Pullable f where
   pull :: f (Box r a) %1-> Box r (f a)
class Pullable2 f where
```

```
  pull2 :: f (Box r a) (Box r b) %1-> Box r (f a b)
```

Separate classes here are defined for unary and binary cases, as working generically over both is tricky in Haskell. Implementing an instance of `Pushable2` for pairs is then:

```
instance Pushable2 (,) where
  push2 (Box (x, y)) = (Box x, Box y)
```

This implementation follows the procedure of Section 4.2. A pair type $A \otimes B$ is handled by pattern matching on `(Box (x, y))` and boxing both fields in the pair `(Box x, Box y)`.

A Haskell programmer may define instances of these type classes for their own data types, but as discussed in Section **??**, this is tedious from a software engineering perspective. Template Haskell is a meta-programming system for Haskell that allows compile-time generation of code, and one of the use cases for Template Haskell is generating boilerplate code **?**. The instances of `Pushable` and `Pullable` for algebraic data types are relatively straightforward, so we implemented procedures that will automatically generate these type class instances for arbitrary user-defined types (though types with so-called "phantom" parameters are currently not supported).

For example, if a programmer wanted to define a `Pushable` instance for the data type of a linear `List a`, they would write:

```
data List a where
  Cons :: a %1-> List a %1-> List a
  Nil  :: List a
$(derivePushable ''List)
```

Here, `derivePushable` is a Template Haskell procedure[3] that takes a name of a user-defined data type and emits a top-level declaration, following the strategy outlined in Section 4.2. For the `List a` data type above, we can walk through

[3]Available online: `https://github.com/granule-project/deriving-distributed-linear-haskell`

the type as `derivePushable` would. Because `List a` has two constructors, a case statement is necessary (our code generator will use a 'case lambda'). This case will have branches for each of the `Cons` and `Nil` constructors—in the body of the former it must box the first field (of type `a`) and recursively apply `push` to the second field (of type `List a`) after boxing it, and in the body of the latter it must simply return `Nil`. The full code generated by `derivePushable ''List` is given below.

```
derivePushable ''List
======>
instance Pushable List where
  push
    = \case
        Box (Cons match_a4BD match_a4BE)
          -> (Cons (Box match_a4BD)) (push (Box match_a4BE))
        Box Nil -> Nil
```

Later, in Section 4.6, we will discuss other combinators and type classes that are useful in Linear Haskell.

The applicability of our proposed deriving method to Haskell shows that it is useful beyond the context of the Granule project. Despite Haskell not having a formal semantics, we believe the same equational reasoning can be applied to show that the properties in Section **??** also hold for these distributive laws in Linear Haskell. As more programming languages adopt type system features similar to Granule and Linear Haskell, we expect that deriving/synthesizing or otherwise generically implementing combinators derived from distributive laws will be increasingly useful.

4.5 Typed-analysis of Consumption in Pattern Matching

This paper's study of distributive laws provides an opportunity to consider design decisions for the *typed analysis of pattern matching* since the operations of Section 4.2 are derived by pattern matching in concert with grading. We compare here the choices made surrounding the typing of pattern matching in four works (1) Granule and its core calculus **?** (2) the graded modal calculus Λ^p of Abel and Bernardy **?** (3) the dependent graded system GRAD of Choudhury et al. **?** and (4) Linear Haskell **?**.

Granule Pattern matching against a graded modality $\Box_r A$ (with pattern $[p]$) in Granule is provided by the PBOX rule (Figure **??**) which triggers typing pattern p 'under' a grade r at type A. This was denoted via the optional grade information $r \vdash\ p : A \ \triangleright\ \Gamma$ which then pushes grading down onto the variables bound within p. Furthermore, it is only under such a pattern that wildcard patterns are allowed ([PWILD]), requiring $0 \sqsubseteq r$, i.e., r can approximate 0 (where 0 denotes weakening). None of the other systems considered here have such a facility for weakening via pattern matching.

For a type A with more than one constructor ($|A| > 1$), pattern matching its constructors underneath an r-graded box requires $1 \sqsubseteq r$. For example, eliminating sums inside an r-graded box $\Box_r(A \oplus B)$ requires $1 \sqsubseteq r$ as distinguishing inl or inr constitutes a *consumption* which reveals information (i.e., pattern matching on the 'tag' of the data constructors). By contrast, a type with only one constructor cannot convey any information by its constructor and so matching on it is not counted as a consumption: eliminating $\Box_r(A \otimes B)$ places no requirements on r. The idea that unary data types do not incur consumption (since no information is conveyed by its constructor) is a refinement here to the original Granule paper as described by Orchard et al. **?**, which for [PCON] had only the premise $1 \sqsubseteq r$ rather than $|A| > 1 \implies 1 \sqsubseteq r$ here (although the implementation already reflected this idea).

The Λ^p calculus Abel and Bernardy's unified modal system Λ^p is akin to Granule, but with pervasive grading (rather than base linearity) akin to the coeffect calculus **?** and Linear Haskell **?** (discussed in Section **??**). Similarly to the situation in Granule, Λ^p also places a grading requirement when pattern matching on a sum type, given by the following typing rule in their syntax (**?**, Fig 1, p.4):

$$\frac{\gamma\Gamma \vdash t : A_1 + A_2 \qquad \delta\Gamma, x_i :^q A_i \vdash u_i : C \qquad q \leq 1}{(q\gamma + \delta)\Gamma \vdash \mathsf{case}^q\ t\ \mathsf{of}\ \{\mathsf{inj}_1 x_1 \mapsto u_1; \mathsf{inj}_2 x_2 \mapsto u_2\} : C}\textsc{+-elim}$$

The key aspects here are that variables x_i bound in the case are used with grade q as denoted by the graded assumption $x_i :^q A_i$ in the context of typing u_i and then that $q \leq 1$ which is exactly our constraint that $1 \sqsubseteq r$ (their ordering just runs in the opposite direction to ours). For the elimination of pair and unit types in Λ^p there is no such constraint, matching our idea that arity affects usage, captured in Granule by $|A| > 1 \implies 1 \sqsubseteq r$. Their typed-analysis of patterns is motivated by their parametricity theorems.

GraD The dependent graded type system GRAD of Choudhury et al. also considers the question of how to give the typing of pattern matching on sum types, with a rule in their system (**?**, p.8) which closely resembles the +-ELIM rule for Λ^p:

$$\frac{\Delta; \Gamma_1 \vdash q : A_1 \oplus A_2 \qquad \Delta; \Gamma_2 \vdash b_1 : A_1 \xrightarrow{q} B \qquad \Delta; \Gamma_2 \vdash b_2 : A_2 \xrightarrow{q} B \qquad 1 \leq q}{\Delta; q \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_q\ a\ \mathbf{of}\ b_1; b_2 : B}\textsc{STcase}$$

The direction of the preordering in GRAD is the same as that in Granule but, modulo this ordering and some slight restructuring, the case rule captures the same idea as Λ^p: "both branches of the base analysis *must* use the scrutinee at least once, as indicated by the $1 \leq q$ constraint." (**?**, p.8). Choudhury et al., also provide a heap-based semantics which serves to connect the meaning of grades with a concrete operational model of usage, which then motivates the grading on sum elimination here. In the simply-typed version of GRAD, matching on the components of a product requires that each component is consumed linearly.

Linear Haskell The paper on Linear Haskell by Bernardy et al. **?** has a **case** expression for eliminating arbitrary data constructors, with grading similar to the rules seen above. Initially, this rule is for the setting of a semiring over $\mathcal{R} = \{1, \omega\}$ (described in Section **??**) and has no requirements on the grading to represent the notion of inspection, consumption, or usage due to matching on (multiple) constructors. This is reflected in the current implementation where we can define the following sum elimination:

```
match :: (Either a b) %r -> (a %1 -> c) -> (b %1 -> c) -> c
match (Left x) f _  = f x
match (Right x) _ g = g x

```

However, later when considering the generalisation to other semirings they state that "*typing rules are mostly unchanged with the caveat that* $\texttt{case}_\pi$ *must exclude* $\pi = 0$" (**?**, §7.2, p.25) where π is the grade of the **case** guard. This appears a more coarse-grained restriction than the other three systems, excluding even the possibility of Granule's weakening wildcard pattern which requires $0 \leq \pi$. Currently, such a pattern must be marked as `'Many` in Linear Haskell (i.e., it cannot explain that first projection on a pair does not use the pair's second component). Furthermore, the condition $\pi \neq 0$ does not require that π actually represents a consumption, unlike the approaches of the other three systems. The argument put forward by Abel and Bernardy for their restriction to mark a consumption ($q \leq 1$) for the sake of parametricity is a compelling one, and the concrete model of Choudhury et al. gives further confidence that this restriction captures well an operational model. Thus, it seems there is a chance for fertilisation between the works mentioned here and Linear Haskell's vital work, towards a future where grading is a key tool in the typed-functional programmer's toolbox.

4.6 Deriving Other Useful Structura Combinators

So far we have motivated the use of distributive laws, and demonstrated that they are useful in practice when programming in languages with linear and graded modal types. The same methodology we have been discussing can also be used to derive other useful generic combinators for programming with linear and graded modal types. In this section, we consider two structural combinators, `drop` and `copyShape`, in Granule as well as related type classes for dropping, copying, and moving resources in Linear Haskell.

4.6.1 A Combinator for Weakening ("drop")

The built-in type constants of Granule can be split into those which permit structural weakening C^{w} such as `Int`, `Char`, `String`, and those which do not C^{l} such as `Handle` (file handles) and `Chan` (concurrent channels). Those that permit weakening contain non-abstract values that can in theory be systematically inspected in order to consume them. Granule provides a built-in implementation of `drop` for C^{w} types, which is then used by the following derivation procedure to derive weakening on compound types:

$$[\![C^{w}]\!]^{\Sigma}_{\mathsf{drop}} z = \texttt{drop}\ z$$

$[\![1]\!]^{\Sigma}_{\mathsf{drop}} z =$ <<no parses (char 12): case z of u***nit . -> unit . >>

$$[\![X]\!]^{\Sigma}_{\mathsf{drop}} z = \Sigma(X) z$$

$$[\![A \oplus B]\!]^{\Sigma}_{\mathsf{drop}} z = \frac{\begin{array}{l}(C_i : B_1{}^{q^i_1} \to \ldots \to B_n{}^{q^i_n} \to K\ \vec{A}) \in D \\ \Gamma, x :_r K\ \vec{A}, y^i_1 :_{r \cdot q^i_1} B_1, \ldots, y^i_n :_{r \cdot q^i_1} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\ \vec{A}, y^i_1 :_{s^i_1} B_1, \ldots, y^i_n :_{s^i_n} \\ \exists {s'}^i_j .\ s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j \\ s_i = {s'}^i_1 \sqcup \ldots \sqcup {s'}^i_n \\ |K\ \vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup \ldots \sqcup s_m\end{array}}{\Gamma, x :_r K\ \vec{A} \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y^i_1 \ldots y^i_n \mapsto t_i} \mid (\Delta_1 \sqcup \ldots \sqcup \Delta_m), x :_{(r_1 \sqcup \ldots \sqcup r_m) + (s_1 \sqcup \ldots \sqcup s}}$$

$$[\![A \otimes B]\!]^{\Sigma}_{\mathsf{drop}} z = \frac{\begin{array}{l}(C_i : B_1{}^{q^i_1} \to \ldots \to B_n{}^{q^i_n} \to K\ \vec{A}) \in D \\ \Gamma, x :_r K\ \vec{A}, y^i_1 :_{r \cdot q^i_1} B_1, \ldots, y^i_n :_{r \cdot q^i_1} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\ \vec{A}, y^i_1 :_{s^i_1} B_1, \ldots, y^i_n :_{s^i_n} \\ \exists {s'}^i_j .\ s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j \\ s_i = {s'}^i_1 \sqcup \ldots \sqcup {s'}^i_n \\ |K\ \vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup \ldots \sqcup s_m\end{array}}{\Gamma, x :_r K\ \vec{A} \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y^i_1 \ldots y^i_n \mapsto t_i} \mid (\Delta_1 \sqcup \ldots \sqcup \Delta_m), x :_{(r_1 \sqcup \ldots \sqcup r_m) + (s_1 \sqcup \ldots \sqcup s}}$$

$$[\![\mu X.A]\!]^{\Sigma}_{\mathsf{drop}} z = \textbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : A \multimap 1}_{\mathsf{drop}}\ \textbf{in}\ f\ z$$

Note we cannot use this procedure in a polymorphic context (over type variables α) since type polymorphism ranges over all types, including those which cannot be dropped like C^{l}.

4.6.2 A Combinator for Copying "shape"

The "shape" of values for a parametric data types F can be determined by a function *shape* : $\mathsf{F}A \to \mathsf{F}1$, usually derived when F is a functor by mapping with $A \to 1$ (dropping elements) ?. This provides a way of capturing the size, shape, and form of a data structure. Often when programming with data structures which must be used linearly, we may wish to reason about properties of the data structure (such as the length or "shape" of the structure) but we may not be able to drop the contained values. Instead, we wish to extract the shape but without consuming the original data structure itself.

This can be accomplished with a function which copies the data structure exactly, returning this duplicate along with a data structure of the same shape, but with the terminal nodes replaced with values of the unit type 1 (the 'spine'). For example, consider a pair of integers: `(1, 2)`. Then applying `copyShape` to this pair would yield `((), ()), (1, 2))`. The original input pair is duplicated and returned on the right of the pair, while the left value contains a pair with the same structure as the input, but with values replaced with `()`. This is useful, as it allows us to use the left value of the resulting pair to reason about the structure of the input (e.g., its depth / size), while preserving the original input. This is particularly useful for deriving size and length combinators for collection-like data structures. As with "drop", we can derive such a function automatically:

$$[\![\mathsf{F}\alpha]\!]_{\mathsf{copyShape}} : \mathsf{F}\alpha \multimap \mathsf{F}1 \otimes \mathsf{F}\alpha$$

defined by $[\![A]\!]_{\mathsf{copyShape}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{copyShape}} z$ by an intermediate derivation $[\![A]\!]^{\Sigma}_{\mathsf{copyShape}}$:

$$[\![C^w]\!]^{\Sigma}_{\mathsf{copyShape}} z = (\texttt{<<no parses (char 2): un***it . >>},\ z)$$

$$[\![1]\!]^{\Sigma}_{\mathsf{copyShape}} z = \frac{\begin{array}{l}(C_i : {B_1}^{q_1^i} \to ... \to {B_n}^{q_n^i} \to K\,\vec{A}) \in D \\ \Gamma, x :_r K\,\vec{A}, y_1^i :_{r \cdot q_1^i} B_1, ..., y_n^i :_{r \cdot q_1^i} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\,\vec{A}, y_1^i :_{s_1^i} B_1, ..., \\ \exists {s'}^i_j .\ s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j \\ s_i = {s'}^i_1 \sqcup ... \sqcup {s'}^i_n \\ |K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup ... \sqcup s_m\end{array}}{\Gamma, x :_r K\,\vec{A} \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y_1^i...y_n^i \mapsto t_i} \mid (\Delta_1 \sqcup ... \sqcup \Delta_m), x :_{(r_1 \sqcup ... \sqcup r_m)+(s_1}}$$

$$[\![\alpha]\!]^{\Sigma}_{\mathsf{copyShape}} z = (\texttt{<<no parses (char 3): un***it . >>}, z)$$

$$[\![X]\!]^{\Sigma}_{\mathsf{copyShape}} z = \Sigma(X) z$$

$$[\![A \oplus B]\!]^{\Sigma}_{\mathsf{copyShape}} z = \frac{\begin{array}{l}(C_i : {B_1}^{q_1^i} \to ... \to {B_n}^{q_n^i} \to K\,\vec{A}) \in D \\ \Gamma, x :_r K\,\vec{A}, y_1^i :_{r \cdot q_1^i} B_1, ..., y_n^i :_{r \cdot q_1^i} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\,\vec{A}, y_1^i :_{s_1^i} B_1, ..., \\ \exists {s'}^i_j .\ s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j \\ s_i = {s'}^i_1 \sqcup ... \sqcup {s'}^i_n \\ |K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup ... \sqcup s_m\end{array}}{\Gamma, x :_r K\,\vec{A} \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y_1^i...y_n^i \mapsto t_i} \mid (\Delta_1 \sqcup ... \sqcup \Delta_m), x :_{(r_1 \sqcup ... \sqcup r_m)+(s_1}}$$

$$[\![\mu X.A]\!]^{\Sigma}_{\mathsf{copyShape}} z = \textbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : A \multimap 1 \otimes A}_{\mathsf{copyShape}}\ \textbf{in}\ f\ z$$

The implementation recursively follows the structure of the type, replicating the constructors, reaching the crucial case where a polymorphically type $z : \alpha$ is mapped to `<<no parses (char 3):  (u***nit ., z) >>` in the third equation.

Granule implements both these derived combinators in a similar way to *push*/*pull* providing `copyShape` and `drop` which can be derived for a type `T` via type application, e.g. `drop @T : T → ()` if it can be derived. Otherwise, the type checker produces an error, explaining why `drop` is not derivable at type `T`.

4.6.3 Other Combinators in Linear Haskell

As previously covered in Section **??**, we demonstrated that the *push* and *pull* combinators derived from distributed laws are useful in Haskell with its linear types extension, and we demonstrated that they can be automatically derived using compile-time meta-programming with Template Haskell.

To the best of our knowledge, nothing comparable to the `Pushable` and `Pullable` type classes proposed here has been previously discussed in the literature on Linear Haskell. However, several other type classes have been proposed for inclusion in the standard library to deal with common use cases when programming with linear function types.[4] One of these classes, `Consumable`, roughly corresponds to the `drop` combinator above, while the other two, `Dupable` and `Movable`, are for when a programmer wants to allow a data type to be duplicated or moved in linear code.

```
class Consumable a where
  consume :: a %1-> ()

class Consumable a => Dupable a where
  dup2 :: a %1-> (a, a)

class Dupable a => Movable a where
  move :: a %1-> Ur a

```

The `consume` function is a linear function from a value to unit, whereas `dup` is a linear function from a value to a pair of that same value. The `move` linear function

[4]See the `linear-base` library: `https://github.com/tweag/linear-base`.

maps `a` to `Ur a`, where `Ur` is the "unrestricted" modality. Thus, `move` can be used to implement both `consume` and `dup2`:

```
        case move x of {Ur _ -> ()}     -- consume x
        case move x of {Ur x -> x}      -- x
        case move x of {Ur x -> (x, x)} -- dup2 x

```

A 'copy shape' class may also be a useful addition to Linear Haskell in the future.

4.7 Discussion and Conclusion

Section 4.5 considered, in some detail, systems related to Granule and different typed analyses of pattern matching. Furthermore, in applying our approach to both Granule and Linear Haskell we have already provided some detailed comparison between the two. This section considers some wider related work alongside ideas for future work, then concludes the paper.

Generic Programming Methodology The deriving mechanism for Granule is based on the methodology of generic functional programming **?**, where functions may be defined generically for all possible data types in the language; generic functions are defined inductively on the structure of the types. This technique has notably been used before in Haskell, where there has been a strong interest in deriving type class instances automatically. Particularly relevant to this paper is the work on generic deriving **?**, which allows Haskell programmers to automatically derive arbitrary class instances using standard datatype-generic programming techniques as described above. In this paper we opted to rely on compile-time metaprogramming using Template Haskell **?** instead, but it is possible that some of the combinators we describe could be implemented using generic deriving as well, which is future work.

Non-graded Distributive Laws Distributive laws are standard components in abstract mathematics. Distributive laws between categorical structures used for modelling modalities (like monads and comonads) are well explored. For example, Brookes and Geva defined a categorical semantics using monads combined with comonads via a distributive law capturing both intensional and effectful aspects of a program **?**. Power and Watanabe study in detail different ways of combining comonads and monads via distributive laws **?**. Such distributive laws have been applied in the programming languages literature, e.g., for modelling streams of partial elements **?**.

Graded Distributive Laws Gaboardi et al. define families of graded distributive laws for graded monads and comonads **?**. They include the ability to interact the grades, e.g., with operations such as $\Box_{\iota(r,f)}\Diamond_f A \to \Diamond_{\kappa(r,f)}\Box_r A$ between a graded comonad $\Box_r$ and graded monad $\Diamond_f$ where ι and κ capture information about the distributive law in the grades. In comparison, our distributive laws here are more prosaic since they involve only a graded comonad (semiring graded necessity) distributed over a functor and vice versa. That said, the scheme of Gaboardi et al. suggests that there might be interesting graded distributive laws between $\Box_r$ and the indexed types, for example, $\Box_r(\mathsf{Vec}\, n\, A) \to \mathsf{Vec}\,(r * n)\,(\Box_1 A)$ which internally replicates a vector. However, it is less clear how useful such combinators would be in general or how systematic their construction would be. In contrast, the distributive laws explained here appear frequently and have a straightforward uniform calculation.

We noted in Section 4.2 that neither of our distributive laws can be derived over graded modalities themselves, i.e., we cannot derive $\mathit{push} : \Box_r\Box_s A \to \Box_s\Box_r A$. Such an operation would itself be a distributive law between two graded modalities, which may have further semantic and analysis consequences beyond the normal derivations here for regular types. Exploring this is future work, for which the previous work on graded distributive laws can provide a useful scheme for considering the possibilities here. Furthermore, Granule has both graded comonads and graded monads so there is scope for exploring possible graded distributive laws between these in the future following Gaboardi et al. **?**.

In work that is somewhat adjacent to this paper, we define program synthesis procedures for graded linear types **?**. This program synthesis approach can synthesis *pull* distributive laws that are equivalent to the algorithmically derived operations of this paper. Interestingly the program synthesis approach cannot derive the *push* laws though as its core theory has a simplified notion of pattern typing that doesn't capture the full power of Granule's pattern matches as described in this paper.

Conclusions The slogan of graded types is to imbue types with information reflecting and capturing the underlying program semantics and structure. This provides a mechanism for fine-grained intensional reasoning about programs at the type level, advancing the power of type-based verification. Graded types are a burgeoning technique that is gaining traction in mainstream functional programming and is being explored from multiple different angles in the literature. The work described here addresses the practical aspects of applying these techniques in real-world programming. Our hope is that this aids the development of the next generation of programming languages with rich type systems for high-assurance programming.

Chapter 5

An extended synthesis calculus

5.1 Extending the graded-base calculus

We now define a 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 ??, and Granule ? in the original 'linear base' form.

Our target language comprises the λ-calculus extended with a graded necessity modality as well as arbitrary user-defined algebraic data types (ADTs). The syntax of types is given by:

$$\begin{aligned} A, B &::= A^r \rightarrow B \mid K \mid A\,B \mid \Box_r A \mid \mu X.A \mid X && (\textit{types}) \\ K &::= 1 \mid \otimes && (\textit{type constructors}) \end{aligned}$$

where the function space $A^r \rightarrow B$ annotates the input type with a *grade* r drawn from a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ parameterising the calculus (where the pre-ordering also requires that $+$ and $*$ are monotonic with respect to $\sqsubseteq$). Type constructors K include the unit and multiplicative linear product type here but they are extended by user-defined ADTs in the implementation which can be formed by products, coproducts, and recursive types (though we do not given an explicit syntax to coproducts here). The graded necessity modality $\Box_r A$ is similarly annotated by the grade r being an element of the semiring. Recursive types are $\mu X.A$, which are equi-recursive in the core calculus, with type recursion variables X.

The syntax of terms is given as:

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

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

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

Typing judgements have the form $\Gamma \vdash t : A$ assigning a type A to a term t under Γ, where Γ and Δ range over contexts, given by:

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

That is, a context may be empty $\emptyset$ or extended with a *graded* assumption $x :_r A$. Graded assumptions must be used in a term in a way which adheres to the constraints provided by the grade r. Structural exchange is permitted, allowing a context to be arbitrarily reordered.

Given a typing judgment $\Gamma \vdash t : A$ we say that t is both *well typed* and *well*

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

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

$$\frac{(C : {B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A}) \in D}{0 \cdot \Gamma \vdash C : {B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A}}\ \textsc{Con} \qquad \frac{\Gamma \vdash t : A \qquad r \vdash p_i : A \rhd \Delta_i \qquad \Gamma', \Delta_i \vdash t_i : B}{r \cdot \Gamma + \Gamma' \vdash \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; ...; p_n \mapsto t_n : B}$$

Figure 6: Typing rules for GrLang

resourced to highlight the role of grading in accounting for resource use via the semiring information.

Figure 6 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, defined as:

Definition 5.1.1 (Scalar context multiplication)**.**

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

i.e. for a non-empty context Γ, each assumption in Γ has its associated grade scaled by r.

Abstraction (Abs) captures the assumption's grade r onto the function arrow in the conclusion, that is, abstraction binds a variable x which may be used in the body t according to grade r. Application (App) makes use of context addition to combine the contexts used to type the two subterms in the premises of the application rule (providing *contraction*):

Definition 5.1.2 (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:

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

For example, $(x :_1 A, y :_0 A) + x :_1 A = x :_{(1+1)} A, y :_0 A$. Thus, we can see $+$ on contexts as pointwise addition of the contexts via semiring $+$, taking the union of any disjoint parts.

Returning to (App), we see that the context Γ_2 for the argument term is scaled by the grade of the function arrow r, as t_2 is used according to r the resulting term $t_1\, t_2$.

Explicit introduction of graded modalities is achieved via the rule for promotion (Pr). The grade r is propagated to the assumptions in Γ through the scaling of Γ by r. Approximation (Approx) allows a grade r to be converted to another grade s, provided that r *approximates* s. Here, $\sqsubseteq$ is an approximation relation defined as the pre-order relation of r and s's semiring. This relation is occasionally lifted pointwise to contexts: we write $\Gamma \sqsubseteq \Gamma'$ to mean that Γ' overapproximates Γ meaning that for all $(x :_{r'} A) \in \Gamma'$ then $(x :_r A) \in \Gamma$ and $r \sqsubseteq r'$.

Introduction and elimination of data constructors is given by the Con and Case rules respectively, with Case also handling graded modality elimination via pattern matching. For Con, we may type a data constructor C of some data type $K\,\vec{A}$ (with zero or more type parameters represented by $\vec{A}$) if it is present in a global context of data constructors D. Data constructors are closed requiring our context Γ to have zero-use grades, thus we scale Γ by 0. Elimination of data constructors take place via pattern matching over a constructor. Patterns p are typed by the judgement $r \vdash p : A \rhd \Delta$ which states that a pattern p has type A and produces a context of typed binders Δ. The grade r to the left of the turnstile

$$\frac{}{r \vdash p : A \rhd x :_r A}\ \textsc{PVar} \qquad \frac{0 \sqsubseteq r}{r \vdash \_ : A \rhd \emptyset}\ \textsc{PWild}$$

$$\frac{r \cdot s \vdash p : A \rhd \Gamma}{r \vdash [p] : \Box_s A \rhd \Gamma}\ \textsc{PBox} \quad \frac{\begin{array}{c}(C : {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K\,\vec{A}) \in D \\ q_i \cdot r \vdash p_i : B_i \rhd \Gamma_i \qquad |K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq r\end{array}}{r \vdash C\,p_1 \ldots p_n : K\,\vec{A} \rhd \overrightarrow{\Gamma_i}}\ \textsc{PCon}$$

Figure 7: Pattern typing rules of the graded calculus

represents the grade information arising from usage in the context generated by this pattern match. The pattern typing rules are given by Figure 7.

Variable patterns are typed by PVar, which simply produces a singleton context containing an assumption $x :_r A$ from the variable pattern with any grade r. A wildcard pattern $\_$, typed by the PWild rule, is only permissible with grades that allow for weakening, i.e., where $0 \sqsubseteq r$. Pattern matching over data constructors is handled by the PCon rule. A data constructor may have up to zero or more sub-patterns ($p_1 \ldots p_n$), each of which is typed under the grade $q_i \cdot r$ (where q_i is the grade of corresponding argument type for the constructor, as defined in D). Additionally, we have the constraint $|K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq r$ which witnesses the fact that if there is more than one data constructor for the data type (written $|K\,\vec{A}| > 1$), then r must approximate 1 because pattern matching on a data constructor incurs some usage since it reveals information about that constructor.[1] By contrast, pattern matching on a type with only one constructor cannot convey any information by itself and so no usage requirement is imposed. Finally, elimination of a graded modality (often called *unboxing*) takes place via the PBox rule, with syntax $[p]$. Like PCon, this rule propogates the grade information of the box pattern's type s to the enclosed sub-pattern p, yielding a context with the grades $r \cdot s$. One may observe that PBox (and by extension Pr) could be

[1]A discussion of this additional constraint on grades for a case expression is given by **?**, including a comparison with how this manifests in various approaches.

considered as special cases of PCon (and Con respectively), if we were to treat our promotion construct as a data constructor with the type $A^r \to \Box_r A$. We find it helpful to keep explicit modality introduction and elimination distinct from constructors, however, particularly with regard to synthesis.

We conclude with two examples of graded modalities indexed respectively by the semiring of intervals over natural numbers and the {0,1,ω} semiring for expressing intuitionistic linear logic.

Example 5.1.1. We have already seen examples of the natural numbers semiring with discrete ordering $(\mathbb{N}, *, 1, +, 0, \equiv)$, which counts exactly how many times variables are used. We denote this semiring as $\mathbb{N}_{\equiv}$ going forwards. This semiring is less useful in the presence of control-flow, e.g., when considering multiple branches in a case expression, where each branch uses a variable differently. A semiring of natural number intervals **?** is more helpful in expressing this behaviour. An interval is given by a pair of natural numbers $\mathbb{N} \times \mathbb{N}$ written $r...s$. The lower bound of the interval is given by $r \in \mathbb{N}$ and the upper bound by $s \in \mathbb{N}$, with $0 = 0...0$ and $1 = 1...1$. Addition is defined pointwise and multiplication defined as in interval arithmetic, with ordering defined as $r...s \sqsubseteq r'...s' = r' \leq r \wedge s \leq s'$. This semiring allows us to write a function which performs an elimination on a coproduct (assuming $\oplus \in D$) as:

$$\oplus_{elim} : (A^1 \to C)^{0...1} \to (B^1 \to C)^{0...1} \to (A \oplus B)^1 \to C$$
$$\oplus_{elim} = \lambda f.\lambda g.\lambda x.\textbf{case } x \textbf{ of } \text{inl } y \mapsto f\ y \mid \text{inr } z \mapsto g\ z$$

Example 5.1.2. The ! modality can be (almost) recovered via the {0,1,ω} semiring. For this semiring, we define + as $r + s = r$ if $s = 0$, $r + s = s$ if $r = 0$, otherwise ω. Multiplication is $r \cdot 0 = 0 \cdot r = 0$, $r \cdot \omega = \omega \cdot r = \omega$ (where $r \neq 0$), and $r \cdot 1 = 1 \cdot r = r$. Ordering is defined as $0 \sqsubseteq \omega$ and $1 \sqsubseteq \omega$. This semiring allows us to express both linear and non-linear usage of values, with a 1 grade indicating linear use, 0 requires the value be discarded, and ω acting as linear logic's !

and permitting unrestrained use. This is similar to Linear Haskell's multiplicity annotations (although Linear Haskell has no equivalent of a 0 grade, only having `One` and `Many` annotations) **?**. Using this semiring, we can write the k-combinator from the SKI calculus as:

$$k : A^1 \to B^0 \to A$$
$$k = \lambda x.\lambda y.x$$

Note however that some additional restrictions are required on pattern typing to get exactly the behaviour of ! with respect to products **?**. This is an orthogonal discussion and not relevant in the rest of this paper.

Lastly we note that the calculus enjoys admissibility of substitution **?** which is critical in type preservation proofs, and is needed in our proof of soundness for synthesis:

Lemma 5.1.1 (Admissibility of substitution). *Let* $\Delta \vdash t' : A$*, then: If* $\Gamma, x :_r A, \Gamma' \vdash t : B$ *then* $\Gamma + (r \cdot \Delta) + \Gamma' \vdash [t'/x]t : B$

5.2 Synthesis

Having defined the target language, we define the core synthesis calculus, which uses the *additive* approach to resource management (discussed in the Section **??**), with judgements of the form:

$$\Gamma \vdash A \Rightarrow^{+} t;\, \Delta$$

That is, given an input context Γ, for goal type A we can synthesis the term t with the output context Δ describing how variables were used in t. The graded context Δ need not necessarily use all the variables in Γ, nor with exactly the same grades. Instead, the relationship between synthesis and typing is given by the following soundness result (the appendix provides the proof):

Theorem 5.2.1 (Soundness of synthesis)**.** *For all graded typing contexts* Γ *and* Δ*, types* A*, terms* t*, and fixing some pre-ordered semiring* $\mathcal{R}$ *parameterising the calculi, then:*

$$\Gamma \vdash A \Rightarrow^{+} t; \Delta \quad \implies \quad \Delta \vdash t : A$$

i.e. t *has type* A *under context* Δ*, which contains variables with grades reflecting their use in* t*.*

The soundness result on its own does not guarantee that the synthesised program t is *well resourced*: i.e., the grades in Δ may not be approximated by the grades in Γ. For example, a valid judgement (whose more general rule is seen shortly) under semiring $\mathbb{N}_{\equiv}$ is:

$$x :_2 A \vdash A \Rightarrow^{+} x;\ x :_1 A$$

i.e., for goal A, if x has type A in the context then we synthesis x as the result program, regardless of the grades. A synthesis judgement such as this may be part of a larger derivation in which the grades eventually match, i.e., this judgement forms part of a larger derivation which has a further subderivation in which x is used again and thus the total usage for x is eventually 2 as prescribed by the input context. However, at the level of an individual judgement we do not guarantee that the synthesised term is well-resourced. A reasonable *pruning condition* that could be used to assess whether any synthesis judgement is *potentially* well-resourced is $\exists \Delta'.(\Delta + \Delta') \sqsubseteq \Gamma$, i.e., there is some additional usage Δ' (that might come from further on in the synthesis process) that ‘fills the gap’ in resource use to produce $\Delta + \Delta'$ which is overapproximated by Γ. In this example, $\Delta' = x :_1 A$ would satisfy this constraint, explaining that there is some further possible single usage which will satisfy the incoming grade. However, previous work on graded linear types showed that excessive pruning at every step becomes too costly in a general setting **?**. Instead, we apply such pruning more judiciously, only requiring that variable use is well-resourced at the point of synthesising binders. Therefore synthesised closed terms are always well-resourced.

For open terms, the implementation checks that from a user-given top-level goal A for which $\Gamma \vdash A \Rightarrow^{+} t; \Delta$ is derivable then t is only provided as a valid (well-typed and well-resourced) synthesis result if $\Delta \sqsubseteq \Gamma$.

We present the synthesis calculus in stages. Each type former of the core calculus (with the exception of type variables) has two corresponding synthesis rules: a right rule for introduction (labelled R) and a left rule for elimination (labelled L). We frequently apply the algorithmic reading of the judgements, where meta-level terms to the left of $\Rightarrow$ are inputs (i.e., context Γ and goal type A) and terms to the right of $\Rightarrow$ are outputs (i.e., the synthesised term t and the usage context Δ).

Section 5.2.1 focuses primarily on a simply-typed core of synthesis, explaining how each synthesis rule addresses the issue of resource management, and how usage information is conveyed from a rule's input to its output. Section 5.2.2 explains the approach to synthesising recursive programs (and handling recursive data types). Section 5.2.3 briefly explains the handling of polymorphism.

The synthesis calculus is non-deterministic, i.e., for any Γ and A there may be many possible t and Δ such that $\Gamma \vdash A \Rightarrow^{+} t; \Delta$. Section 5.2.5 explains how the core rules are 'reorganised' into a deterministic system via the approach of focusing proof search. Lastly, Section 5.2.6 briefly explains post-synthesis refactoring to produce shorter, easier to read programs.

Whilst we largely present the approach here in abstract terms, via the synthesis judgements, we highlight some implementation choices (e.g., heuristics applied in the algorithmic version of the rules) as given in our implementation into Granule.

5.2.1 Core synthesis rules

Variables For any goal type A, if there is a variable in the context matching this type then it can be synthesised for the goal, given by the terminal rule:

$$\frac{}{\Gamma, x :_r A \vdash A \Rightarrow x \mid 0 \cdot \Gamma, x :_1 A}\ \textsc{Var}$$

Said another way, to synthesise the use of a variable x, we require that x be present in the input context Γ. The output context here then explains that only variable x is used: it consists of the entirety of the input context Γ scaled by grade 0 (using definition 5.1.1), extended with $x :_1 A$, i.e. a single usage of x as denoted by the 1 element of the semiring. Maintaining this zeroed Γ in the output context simplifies subsequent rules by avoiding excessive context membership checks.

The Var rule permits the synthesis of terms which may not be well-resourced, e.g., if $r = 0$, the rule still synthesises a use of x. This is locally ill-resourced, but is acceptable at the global level as we check that an assumption has been used correctly in the rule where the assumption is bound. This does leads us to consider some branches of synthesis that are guaranteed to fail: at the point of synthesising a usage of a variable in the additive scheme, isolated from information about how else the variable is used, there is no way of knowing if such a usage will be permissible in the final synthesised program. However, it also reduces the amount of intermediate theorems that need solving, which can significantly effect performance as shown by **?**, especially since the variable rule is applied very frequently.

Functions Synthesis of programs from function types is handled by the Absand Apprules, which synthesise abstraction and application terms, respectively. An abstraction is synthesised like so:

$$\frac{\Gamma, x :_q A \vdash B \Rightarrow t \mid \Delta, x :_r A \qquad r \sqsubseteq q}{\Gamma \vdash A^q \to B \Rightarrow \lambda x.t \mid \Delta}\ \textsc{Abs}$$

Reading the rule bottom up, to synthesise a term of type $A^q \to B$ in context Γ we first extend the context with a fresh variable assumption $x :_q A$ and synthesise a term of type B that will ultimately become the body of the function. The type $A^q \to B$ conveys that A must be used according to q in our term for B. The fresh variable x is passed to the premise of the rule using the grade of the binder: q. The x must then be used to synthesise a term t with q usage. In the premise, after synthesising t we obtain an output context $\Delta, x :_r A$. As mentioned, the Var rule ensures that x is present in this context, even if it was not used in the synthesis of t (for example if $q = 0$). The rule ensures the usage of bound term (r) in t does not violate the input grade q via the requirement that $r \sqsubseteq q$ i.e. that r *approximates* q. If met, Δ becomes the output context of the rule's conclusion.

The counterpart to abstraction synthesises an application from the occurrence of a function in the context (a left rule):

$$\frac{\begin{array}{c}\Gamma, x_1 :_{r_1} A^q \to B, x_2 :_{r_1} B \vdash C \Rightarrow t_1 \mid \Delta_1, x_1 :_{s_1} A^q \to B, x_2 :_{s_2} B \\ \Gamma, x_1 :_{r_1} A^q \to B \vdash A \Rightarrow t_2 \mid \Delta_2, x_1 :_{s_3} A^q \to B\end{array}}{\Gamma, x_1 :_{r_1} A^q \to B \vdash C \Rightarrow [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + s_2 \cdot q \cdot \Delta_2), x_1 :_{s_2 + s_1 + (s_2 \cdot q \cdot s_3)} A^q \to B}\ \textsc{App}$$

Reading bottom up again, the input context contains an assumption with a function type $x_1 :_{r_1} A^q \to B$. We may attempt to use this assumption in the synthesis of a term with the goal type C, by applying some argument to it. We do this by synthesising the argument from the input type of the function A, and then binding the result of this application as an assumption of type B in the synthesis of C. This is decomposed into two steps corresponding to the two premises (though in the implementation the first premise is considered first):

1. The first premise synthesises a term t_1 from the goal type C under the assumption that the function x_1 has been applied and its result is bound to x_2. This placeholder assumption is bound with the same grade as x_1.

2. The second premise synthesises an argument t_2 of type A for the function x_1. In the implementation, this synthesis step occurs only after a term t_1 is found for the goal C as a heuristic to avoid possibly unnecessary work if no term can be synthesised for C anyway.

In the conclusion of the rule, a term is synthesised which substitutes in t_1 the result placeholder variable x_2 for the application $x_1\,t_2$.

The first premise yields an output context $\Delta_1, x_1 :_{s_1} A^q \rightarrow B, x_2 :_{s_2} B$. The output context of the conclusion is obtained by taking the context addition of Δ_1 and $s_2 \cdot q \cdot \Delta_2$. The output context Δ_2 is first scaled by q since t_2 is used according to q when applied to x_1 (as per the type of x_1). We then scale this again by s_2 which represents the usage of the entire application $x_1\,t_2$ inside t_1.

The output grade of x_1 follows a similar pattern since this rule permits the re-use of x_1 inside both premises of the application (which differs from Hughes and Orchard's treatment of synthesis in a linear setting). As x_1's input grade r_1 may permit multiple uses both inside the synthesis of the application argument t_2 and in t_1 itself, the total usage of t_1 across both premises must be calculated. In the first premise x_1 is used according to s_1, and in the second according to s_3. As with Δ_2, we take the semiring multiplication of s_3 and q and then multiply this by s_2 to yield the final usage of x_1 in t_2. We then add this to $s_2 + s_1$ to yield the total usage of x_1 in t_1.

Data types The synthesis of introduction forms for data types is handled by the CONrule:

$$\frac{\begin{array}{l}(C : {B_1}^{q_1} \rightarrow ... \rightarrow {B_n}^{q_n} \rightarrow K\,\vec{A}) \in D \\ \Gamma \vdash B_i \Rightarrow t_i \mid \Delta_i\end{array}}{\Gamma \vdash K\,\vec{A} \Rightarrow C\ t_1 \ldots t_n \mid 0 \cdot \Gamma + (q_1 \cdot \Delta_1) + \ \ldots\ + (q_n \cdot \Delta_n)}\ \textsc{Con}$$

where D is the set of data constructors in global scope, e.g., coming from ADT definitions, but including here products with $(,) : A^1 \rightarrow B^1 \rightarrow A \otimes B$ and $Unit : \mathsf{1}$.

For a goal type $K\,\vec{A}$ where K is a data type with zero or more type arguments (denoted by the vector $\vec{A}$), then a constructor term $C\ t_1 \ldots t_n$ for $K\,\vec{A}$ is synthesised, This requires a sub-term to be synthesised for each of the constructor's arguments t_i in the second premise (which is repeated for each argument type B_i), yielding output contexts Δ_i. The output context for the rule's conclusion is obtained by performing a context addition across all the output contexts generated from the premises, where each context Δ_i is scaled by the corresponding grade q_i from the data constructor in D capturing the fact that each argument t_i is used according to q_i.

The dual of the above, constructor elimination, synthesises **case** statements with branches pattern matching on each data constructor of the target data type $K\,\vec{A}$, with various associated constraints on grades which require some careful explanation:

$$\frac{\begin{array}{l}(C_i : {B_1}^{q_1^i} \rightarrow ... \rightarrow {B_n}^{q_n^i} \rightarrow K\,\vec{A}) \in D \\ \Gamma, x :_r K\,\vec{A}, y_1^i :_{r \cdot q_1^i} B_1, ..., y_n^i :_{r \cdot q_1^i} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\,\vec{A}, y_1^i :_{s_1^i} B_1, ..., y_n^i :_{s_n^i} B_n \\ \exists {s'}_j^i .\ s_j^i \sqsubseteq {s'}_j^i \cdot q_j^i \sqsubseteq r \cdot q_j^i \\ s_i = {s'}_1^i \sqcup ... \sqcup {s'}_n^i \\ |K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup ... \sqcup s_m\end{array}}{\Gamma, x :_r K\,\vec{A} \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y_1^i ... y_n^i \mapsto t_i} \mid (\Delta_1 \sqcup ... \sqcup \Delta_m), x :_{(r_1 \sqcup ... \sqcup r_m) + (s_1 \sqcup ... \sqcup s_m)} K\,\vec{A}}\ \textsc{Case}$$

where $1 \leq i \leq m$ is used to index the data constructors of which there are m (i.e., $m = |K\,\vec{A}|$) and $1 \leq j \leq n$ is used to index the arguments of the i^{th} data constructor ; for brevity, the rule focuses on the case of n-ary data constructors where $n > 0$.

As with constructor introduction, the relevant data constructors are retrieved from the global context D in the first premise. A data constructor type is a function type from the constructor's arguments $B_1 \ldots B_n$ to a type constructor applied to zero or more type parameters $K\,\vec{A}$. However, in the case of nullary data constructors (e.g., for the unit type), the data constructor type is simply the type constructor's type with no arguments. For each data constructor C_i, we synthesise a term t_i from the result type of the data constructor's type in D, binding the data constructor's argument types as fresh assumptions to be used in the synthesis of t_i.

To synthesise the body for each branch i, the arguments of the data constructor are bound to fresh variables in the premise, with the grades from their respective argument types in D multiplied by the r. This follows the pattern typing rule for constructors; a pattern match under some grade r must bind assumptions that have the capability to be used according to r.

The assumption being eliminated $x :_r K\,\vec{A}$ is also included in the premise's context (as in App) as we may perform additional eliminations on the current assumption subsequently if the grade r allows us. If successful, this will yield both a term t_i and an output context for the pattern match branch. The output context can be broken down into three parts:

1. Δ_i contains any assumptions from Γ were used to construct t_i
2. $x :_{r_i} K\,A$ describes how the assumption x was used
3. $y_1^i :_{s_1^i} B_1, ..., y_n^i :_{s_n^i} B_n$ describes how each assumption y_j^i bound in the pattern match was used in t_i.

This leaves the question of how we calculate the final grade to attribute to x in the output context of the rule's conclusion. For each bound assumption, we generate a fresh grade variable ${s'}^i_j$ which represents how that variable was used in t_i after factoring out the multiplication by r. This is done via the constraint in the third premise that $\exists {s'}^i_j .\, s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j$. The join of each ${s'}^i_j$ (for each assumption) is then taken to form a grade variable s_i which represents the total usage of x for this branch that arises from the use of assumptions which were bound via the pattern match (i.e. not usage that arises from reusing x explicitly inside t_i). For the output context of the conclusion, we then take the join of output context from the constructors used. This is extended with the original x assumption with the output grade consisting of the join of each r_i (the usages of x directly in each branch) plus the join of each s_i (the usages of the assumptions that were bound from matching on a constructor of x).

Example 5.2.1 (Example of **case** synthesis)**.** Consider two possible synthesis results as:

$$x :_s 1 \oplus A, y :_r A, z :_r A \vdash A \Rightarrow^+ z;\ x :_0 1 \oplus A, y :_0 A, z :_1 A \tag{1}$$

$$x :_s 1 \oplus A, y :_r A \vdash A \Rightarrow^+ y;\ x :_0 1 \oplus A, y :_1 A \tag{2}$$

We will plug these into the rule for generating case expressions as follows where in the following instead of using the above concrete grades we have used the abstract form of the rule (the two will be linked by equations after):

$$\dfrac{\begin{array}{c} \mathsf{Just} : A^1 \to A \oplus 1 \in D \qquad\qquad \mathsf{Nothing} : 1^1 \multimap A \oplus 1 \in D \\ (1) \qquad x :_s 1 \oplus A, y :_r A, z :_{r \cdot q_1} A \vdash A \Rightarrow^+ z;\ x :_0 1 \oplus A, y :_0 A, z :_{s_1} A \\ (2) \qquad x :_s 1 \oplus A, y :_r A \vdash A \Rightarrow^+ y;\ x :_0 1 \oplus A, y :_1 A \\ \exists s'_1 .\, s_1 \sqsubseteq s'_1 \cdot q_1 \sqsubseteq r \cdot q_1 \qquad\qquad s = s_1 \end{array}}{y :_1 A, x :_r 1 \oplus A \vdash A \Rightarrow^+ (\textbf{case}\ x\ \textbf{of}\ \mathsf{Just}\, z \to z; \mathsf{Nothing} \to y);\ x :_{(0 \sqcup 0)+s} 1 \oplus A, y :_{0 \sqcup 1} A}\ \textsc{Case}$$

Thus, to unify (1) and (2) with the rule format we have that $s_1 = 1$ and $q_1 = 1$. Applying these two equalities as rewrites to the remaining constraint, we have that:

$$\exists s'_1 .\, 1 \sqsubseteq s'_1 \cdot 1 \sqsubseteq r \cdot 1 \qquad \Longrightarrow \qquad \exists s'_1 .\, 1 \sqsubseteq s'_1 \sqsubseteq r$$

These constraints can be satisfied with the natural-number intervals semiring where $y :_{0..1} A$

Deep pattern matching, over nested data constructors, is handled via inductively applying the Caserule but with a post-synthesis refactoring procedure substituting the pattern match of the inner case statement into the outer pattern match (Section 5.2.6). For example,

$$\textbf{case}\ x\ \textbf{of}\ (y_1, y_2) \rightarrow \textbf{case}\ y_1\ \textbf{of}\ (z_1, z_2) \rightarrow z_2$$

becomes a single **case** with nested pattern matching, simplifying the synthesised program:

$$\textbf{case}\ x\ \textbf{of}\ ((z_1, z_2), y_2) \rightarrow z_2$$

Graded modalities Graded modalities are introduced and eliminated explicitly through the Boxand Unboxrules, respectively. In Box, we synthesise a promotion $[t]$ for some graded modal goal type $\Box_r A$:

$$\frac{\Gamma \vdash A \Rightarrow t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow [t] \mid r \cdot \Delta}\ \textsc{Box}$$

In the premise, we synthesise from A, yielding the subterm t and an output context Δ. In the conclusion, Δ is scaled by the grade of the goal type r: as $[t]$ must use t as r requires.

As with data constructors, grade elimination (*unboxing*) takes place via pattern matching in **case**:

$$\frac{\begin{array}{l}\Gamma, y :_{r \cdot q} A, x :_r \Box_q A \vdash B \Rightarrow t \mid \Delta, y :_{s_1} A, x :_{s_2} \Box_q A \\ \exists s_3 .\ s_1 \sqsubseteq s_3 \cdot q \sqsubseteq r \cdot q\end{array}}{\Gamma, x :_r \Box_q A \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ [y] \to t \mid \Delta, x :_{s_3 + s_2} \Box_q A}\ \textsc{Unbox}$$

To eliminate the assumption x, which has the graded modal type $\Box_q A$, we bind a fresh assumption in the synthesis of the premise: $y :_{r_1 \cdot q} A$. This assumption is graded with $r_1 \cdot q$: the grade from the assumption's type multiplied by the grade of the assumption itself. As with previous elimination rules, x is rebound in the rule's premise. Some term t is then synthesised resulting in the output context $\Delta, y :_{s_1} A, x :_{s_2} \Box_q A$, where s_1 and s_2 describe how y and x were used in t. The second premise ensures that the usage of y does not exceed the binding grade $r_1 \cdot q$. For the output context of the conclusion, we simply remove the bound y from Δ and add x, with the grade $s_2 + s_3$: representing the total usage of x in t.

5.2.2 Recursion

Synthesis permits recursive definitions, as well as programs which may make use of calls to functions from a user-supplied context of function definitions in scope (see 5.2.4).

Synthesis of non-recursive function applications may take place arbitrarily, however, synthesising a recursive function definition application requires more care. In order to ensure that a synthesised programs terminates, we only permit synthesis of terms which are *structurally recursive*, i.e., those which apply the recursive definition to a subterm of the function's inputs **?**.

Synthesis rules for recursive data structures (μ-types) are fairly straightforward:[2]

$$\frac{\Gamma \vdash A[\mu X.A/X] \Rightarrow t \mid \Delta}{\Gamma \vdash \mu X.A \Rightarrow t \mid \Delta}\,\mu_{\mathrm{R}} \qquad \frac{\Gamma, x :_r A[\mu X.A/X] \vdash B \Rightarrow t \mid \Delta}{\Gamma, x :_r \mu X.A \vdash B \Rightarrow t \mid \Delta}\,\mu_{\mathrm{L}}$$

This μ_{R} rule states that to synthesise a recursive data structure of type $\mu X.A$, we must be able to synthesise A with $\mu X.A$ substituted for the recursion variables X in A. For example, if we wish to synthesise a list data type `List a` with constructors `Nil` and `Cons a (List a)`, then when choosing the `Cons` constructor in the μ_{R} rule, the type of this constructor requires us to re-apply the μ_{R} rule, to synthesise the recursive part of `Cons`. Elimination of a recursive data structure may be synthesised using the μ_{L} rule. In this rule, we have some recursive data type $\mu X.A$ in our context which we may wish to pattern match on via the $\mathrm{C_L}$ rule. To do this, the assumption is bound in the premise with the type A, substituting $\mu X.A$ for the recursion variables X in A.

Recursive data structures present a challenge in the implementation. For our list data type, how do we prevent our synthesis tool from simply applying the μ_{L} rule, followed by the $\mathrm{C_L}$ rule on the `Cons` constructor ad infinitum? We resolve this issue using an *iterative deepening* approach to synthesis similar to the approach used by Myth **?**. Programs are synthesised with elimination (and introduction) forms of constructors restricted up to a given depth. If no program is synthesised within these bounds, then the depth limits are incremented. Combined with focusing (see section 5.2.5), this provides the basis for an efficient implementation of the above rules.

[2]Though μ types are equi-recursive, we make explicit the synthesis rules here which maps more closely to the implementation where iterative deepening information needs to be tracked at the points of using μ_{L} and μ_{R}.

5.2.3 Polymorphism

Granule features rank-0 polymorphism à la ML, i.e., via *type schemes* which have universal quantification of type variables only at the top-level of a type (though for brevity we elided polymorphism from the core calculus here). Programs in our approach can be synthesised from a polymorphic type scheme, treating universal type variables quantified at the top-level of our goal type as logical atoms which cannot be unified with and are only equal to themselves.

When a synthesising an introduction/elimination form of a polymorphic constructor or an application involving a polymorphic top-level function definition, the constructor/definition is instantiated with fresh unification variables which are then used as the basis for further synthesis. Substitution and unification are treated in a standard way. For example, in the following we have a polymorphic function we want to use to synthesise a monomorphic function:

```
flip : ∀ c d  . (c, d) %1 → (d, c)
flip (x, y) = (y, x)
f : (Int, Int) %1 → (Int, Int)
f x = ? -- synthesis to flip x trivially
```

To synthesis the application `flip x` here, the type scheme for `flip` is instantiated with fresh unification variables c', d', yielding `(c', d') %1 → (d', c')` from which c' and d' are then unified with `Int` in an application of a Var rule.

5.2.4 Input-output examples

When specifying the synthesis context of top-level definitions, the user may also supply a series of input-output examples showcasing desired behaviour. Our approach to examples is deliberately naïve; we evaluate a fully synthesised candidate program against the inputs and check that the results match the corresponding outputs. Unlike many sophisticated example-driven synthesis tools, the examples here do not themselves influence the search procedure, and are used solely to allow the user to clarify their intent. This lets us consider the effectiveness of basing the search primarily around the use of grade information. An approach to synthesis of resourceful programs with examples closely integrated into the search as well is further work.

We augmented the Granule language with first-class syntax for specifying input-output examples, both as a feature for aiding synthesis but also for aiding documentation that is type checked (and therefore more likely to stay consistent with a code base as it evolves). Synthesis specifications are written in Granule directly above a program hole (written using `?`) using the `spec` keyword. The input-output examples are then listed per-line.

```
tail : ∀ a . List a %0..1 → List a
spec
  tail (Cons 1 Nil) = Nil;
  tail (Cons 1 (Cons 2 Nil)) = Cons 2 Nil;
tail = ?
```

Any synthesised term must then behave according to the supplied examples. This `spec` structure can also be used to describe additional synthesis components that the user wishes the tool to make use of. These components comprise a list of in-scope definitions separated by commas. The user can choose to annotate each component with a grade, describing the required usage in the synthesised term. This defaults to a 1 grade if not specified. For example, the specification for a function which returns the length of a list would look something like:

```
length : ∀ a . List a %0..∞. → N
spec
    length Nil = Z;
    length (Cons 1 Nil) = S Z;
    length (Cons 1 (Cons 1 Nil)) = S (S Z);
    length %0..∞.
length = ?
```

with the following resulting program produced by our synthesis algorithm (on average in about 400ms on a standard laptop, see Section **??** where this is one of the benchmarks for evaluation):

```
length Nil = Z;
length (Cons y z) = S (length z)
```

5.2.5 Focusing proof search

The calculus presented above serves as a starting point for implementing a synthesis algorithm in Granule. However, at the moment the rules are highly non-deterministic with regards the order in which they may be applied. For example, after applying a $\rightarrow R$ rule, we may choose to apply any of the elimination rules before applying an introduction rule for the goal type. This leads to us exploring a large number of redundant search branches which can be avoided through the application of a technique known as *focusing* **?**. Focusing is a tool from linear logic proof theory based on the idea that some rules are invertible, i.e., whenever the conclusion of the rule is derivable, then so are the premises. In other words, the order in which we apply invertible rules doesn't matter. By fixing a particular ordering on the application of invertible rules, we eliminate much of the non-determinism that arises from trying branches which differ only in the order in which invertible rules are applied. We briefly outline focusing and how we apply it.

We begin by augmenting our previous synthesis judgement with an additional context:

$$\Gamma; \Omega \vdash A \Rightarrow t \mid \Delta$$

Unlike Γ and Δ, Ω is an *ordered* context. Using the terminology of Pfenning, we refer to rules that are invertible as *asynchronous* and rules that are not as *synchronous* **?**. The intuition here is that asynchronous rules can be applied eagerly, while the non-invertible synchronous rules require us to *focus* on a particular part of the judgement: either on the assumption (if we are in an elimination rule) or on the goal (for an introduction rule). When focusing we apply a chain of synchronous rules, until we either reach a position where no rules may be applied (at which point the branch terminates), we have synthesised a term for our goal, or we have exposed an asynchronous connective at which point we switch back to applying asynchronous rules.

We divide our synthesis rules into four categories, each with their own judgement form, refining the focusing judgement above with an arrow indicating which part of the judgement is currently in focus. An $\Uparrow$ indicates an asynchronous phase, and $\Downarrow$ a synchronous (focused) phase. The location of the arrow in the judgement indicates whether we are in an introduction or elimination phase:

1. Right Async: $\rightarrow$ R rule with the judgement $\Gamma; \Omega \vdash A \Uparrow\, \Rightarrow t \mid \Delta$
2. Left Async: $\mathrm{C_L}$, $\square_\mathrm{L}$ rules with the judgement $\Gamma; \Omega \Uparrow\, \vdash A \Rightarrow t \mid \Delta$
3. Right Sync: $\mathrm{C_R}$, $\square_\mathrm{R}$ rules with the judgement $\Gamma; \Omega \vdash A \Downarrow\, \Rightarrow t \mid \Delta$
4. Left Sync: $\rightarrow$ L rule with the judgement $\Gamma; \Omega \Downarrow\, \vdash A \Rightarrow t \mid \Delta$

We find it helpful to view focusing in terms of a finite state machine, as given in figure 8. States comprise the four phases of focusing, plus two additional states, Focus, and Var. Edges are then the synthesis rules that direct the transition between focusing phases. The transitions between these focusing phases are handled by dedicated focusing rules for each transition. For the asynchronous phases, the $\Uparrow_R / \Uparrow_L$ handle the transition between right to left phases, and left to focusing phases, respectively. Conversely, the $\Downarrow R$ rule deals with the transition from a right synchronous phase back to a right asynchronous phase, with the $\Downarrow L$ rule likewise transitioning to a left asynchronous phase. Depending on the current phase of focusing, these rules consider the goal type, the assumption currently being focused on's type, the size of Ω, and the current level of iterative deepening to decide whether to transition between focusing phases. For brevity, we elide the full details here.

Lemma 5.2.2 (Soundness of focusing for graded-base synthesis)**.** *For all contexts* Γ*,* Ω *and types* A*:*

1. *Right Async* : $\Gamma; \Omega \vdash A \Uparrow \Rightarrow t \mid \Delta \quad \Longrightarrow \quad \Gamma, , \Omega \vdash A \Rightarrow^{+} t;\ \Delta$
2. *Left Async* : $\Gamma; \Omega \Uparrow \vdash B \Rightarrow t \mid \Delta \quad \Longrightarrow \quad \Gamma, , \Omega \vdash B \Rightarrow^{+} t;\ \Delta$
3. *Right Sync* : $\Gamma; \emptyset \vdash A \Downarrow\ \Rightarrow t \mid\ \Delta \quad \Longrightarrow \quad \Gamma \vdash A \Rightarrow^{+} t;\ \Delta$
4. *Left Sync* : $\Gamma; x :_r A \Downarrow \vdash B \Rightarrow t \mid \Delta \quad \Longrightarrow \quad \Gamma, x :_r A \vdash B \Rightarrow^{+} t;\ \Delta$
5. *Focus Right* : $\Gamma; \emptyset \vdash B \Rightarrow t \mid\ \Delta \quad \Longrightarrow \quad \Gamma \vdash B \Rightarrow^{+} t;\ \Delta$
6. *Focus Left* : $\Gamma, x :_r A; \emptyset \vdash B \Rightarrow t \mid\ \Delta \quad \Longrightarrow \quad \Gamma, x :_r A \vdash B \Rightarrow^{+} t;\ \Delta$

i.e. t *has type* A *under context* Δ*, which contains variables with grades reflecting their use in* t*. The appendix provides the proof.*

5.2.6 Post-synthesis refactoring

A synthesised term often contains some artefacts of the fact that it was constructed automatically. The structure of our synthesis rules means aspects of our synthesised programs are unrepresentative in some stylistic ways of the kind of programs functional programmers typically write. We consider three examples of these below, and show how we apply a refactoring procedure to any synthesised term to rewrite them in a more idiomatic style.

Abstractions A function definition synthesised from a function type will take the form of a sequence of nested abstractions which bind the function's arguments, with the sub-term of the innermost abstraction containing the function body, e.g.

```
k : ∀ {a b : Type} . a %1 → b %0 → a
k = λx → λy → x
```

In most cases, a programmer would write a function definition as a series of equations with the function arguments given as patterns. Our refactoring procedure collects the outermost abstractions of a synthesised term and transforms them into equation-level patterns with the innermost abstraction body forming the equation body:

```
k : ∀ {a b : Type} . a %1 → b %0 → a
k x y = x
```

Case statements Recall that the C_L binds a data constructor's patterns as a series of variables. Synthesising a pattern match over a nested data structure therefore yields a term such as:

```
case x of
  C₁ y →
    case y of
      D₁ z → ...
      D₂ z → ...
  C₂ y →
    case y of
      D₁ z → ...
      D₂ z → ...
```

which would be rather unnatural for a programmer to write. Nested case statements are therefore folded together to yield a single case statement which pattern matches over all combination of patterns from each statement. The above cases are then transformed into the much more compact and readable single case:

```
case x of
```

```
    C₁ (D₁ z) → ...
    C₁ (D₂ z) → ...
    C₂ (D₁ z) → ...
    C₂ (D₂ z) → ...
```

Furthermore, pattern matches over a function's arguments in the form of case statements are refactored such that a new function equation is created for each unique combination of pattern match. In this way, a refactored program should only contain case statements that arise from pattern matching over the result of an application.

```
neg : Bool %1 → Bool %1
neg x = case x of
          True → False;
          False → True
```

is refactored into:

```
neg : Bool %1 → Bool %1
neg True True  = False;
neg False True = True;
```

The exception to this is where the scrutinee of a case statement is re-used inside one of the case branches, in which case refactoring would cause us to throw away the binding of the scrutinee's name and so it cannot be folded into the head pattern match, for example:

```
last : ∀ a . (List a) %0..∞ → Maybe a
spec
    last Nil = Nothing;
    last (Cons 1 Nil) = Just 1;
    last (Cons 1 (Cons 2 Nil)) = Just 2;
    last %0..∞
last Nil = Nothing;
last (Cons y z) =
    (case z of
      Nil → Just y;
      Cons u v → last z)
```

Unboxing An unboxing term is synthesised via the $\square_L$ rule as a case statement which pattern matches over a box pattern, yielding an assumption with the grade's usage. Such terms can also be refactored both into function equations and to avoid nested case statements. For example, we may write the k combinator example above using an explicit graded modality:

```
k : ∀ {a b : Type} . a %1 → b [0] → a
k x y = case y of [z] → x
```

which we can then refactor into

```
k : ∀ {a b : Type} . a %1 → b [0] → a
k x [z] = z
```

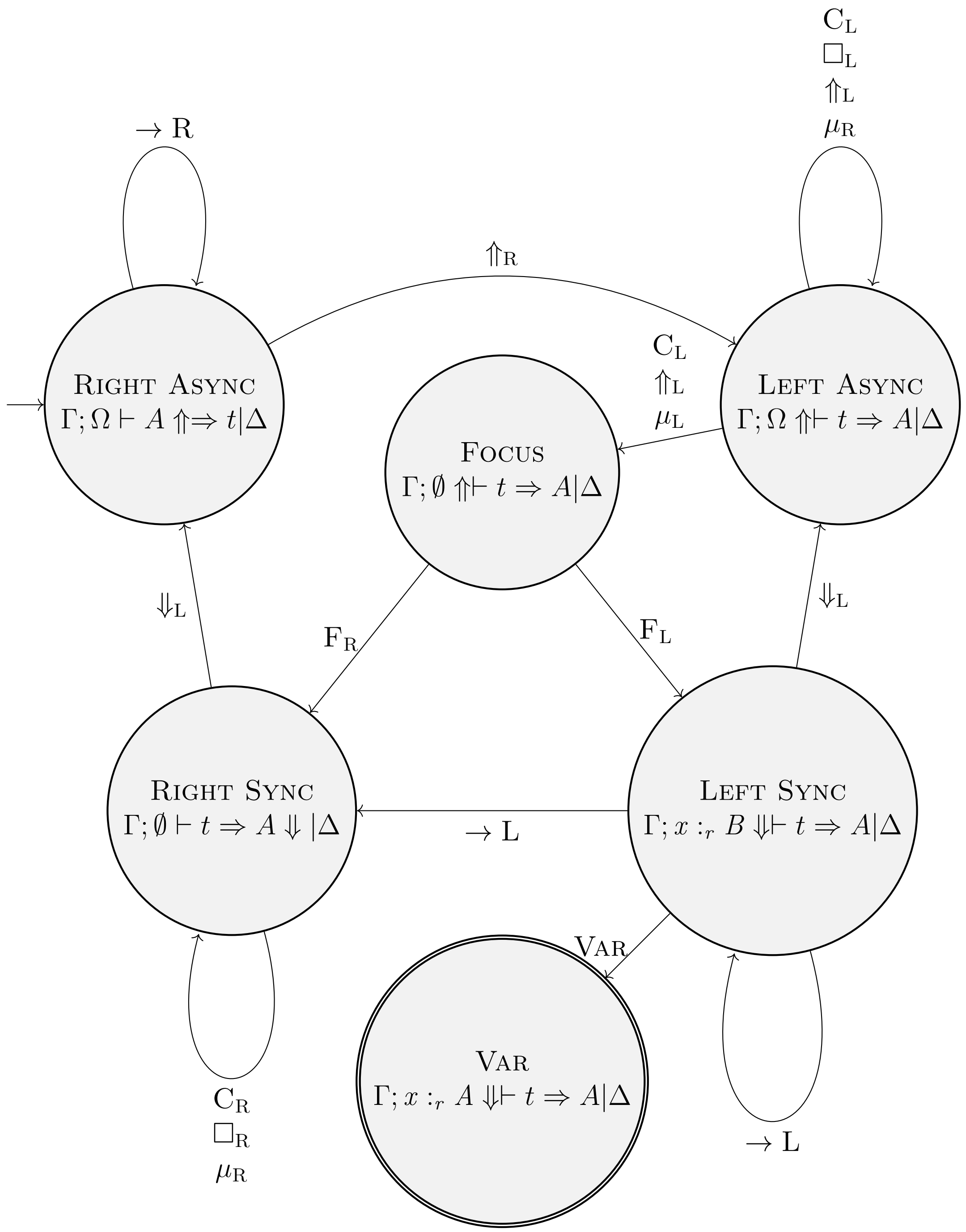

$\to$ R
C_L
$\Box_L$
$\Uparrow_L$
μ_R
$\Uparrow_R$
Right Async
$\Gamma; \Omega \vdash A \Uparrow\Rightarrow t|\Delta$
C_L
$\Uparrow_L$
μ_L
Left Async
$\Gamma; \Omega \Uparrow\vdash t \Rightarrow A|\Delta$
Focus
$\Gamma; \emptyset \Uparrow\vdash t \Rightarrow A|\Delta$
$\Downarrow_L$
$\Downarrow_L$
F_R
F_L
Right Sync
$\Gamma; \emptyset \vdash t \Rightarrow A \Downarrow |\Delta$
Left Sync
$\Gamma; x :_r B \Downarrow\vdash t \Rightarrow A|\Delta$
$\to$ L
Var
Var
$\Gamma; x :_r A \Downarrow\vdash t \Rightarrow A|\Delta$
C_R
$\Box_R$
μ_R
$\to$ L

Figure 8: Focusing State Machine

Chapter 6

Conclusion

Appendix A

Collected synthesis rules

A.1 Collected rules of the linear-base synthesis calculi

A.1.1 Collected rules of the subtractive calculus

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^{-} x \mid \Gamma}\ \textsc{LinVar}^{-} \qquad \frac{\exists s.\, r \sqsubseteq s + 1}{\Gamma, x :_{r} A \vdash A \Rightarrow^{-} x \mid \Gamma, x :_{s} A}\ \textsc{GrVar}^{-}$$

$$\frac{\Gamma, x :_{s} A, y : A \vdash B \Rightarrow^{-} t \mid \Delta, x :_{s'} A \qquad y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s + 1}{\Gamma, x :_{r} A \vdash B \Rightarrow^{-} [x/y]t \mid \Delta, x :_{s'} A}\ \textsc{der}^{-}$$

$$\frac{\Gamma, x : A \vdash B \Rightarrow^{-} t \mid \Delta \quad x \notin |\Delta|}{\Gamma \vdash A \multimap B \Rightarrow^{-} \lambda x.t \mid \Delta}\ \multimap_{R}^{-}$$

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1 \vdash A \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{-} [(x_1\, t_2)/x_2]t_1 \mid \Delta_2}\ \multimap_{L}^{-}$$

$$\frac{\Gamma \vdash A \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash \Box_{r} A \Rightarrow^{-} [t] \mid \Gamma - r \cdot (\Gamma - \Delta)}\ \Box_{R}^{-} \quad \frac{\Gamma, x_2 :_{r} A \vdash B \Rightarrow^{-} t \mid \Delta, x_2 :_{s} A \qquad 0 \sqsubseteq s}{\Gamma, x_1 : \Box_{r} A \vdash B \Rightarrow^{-} \textbf{let}\, [x_2] = x_1\, \textbf{in}\, t \mid \Delta}\ \Box_{L}^{-}$$

$$\frac{\Gamma \vdash A \Rightarrow^{-} t_1 \mid \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{-} (t_1, t_2) \mid \Delta_2}\ \otimes_{R}^{-}$$

A.1.2 Collected rules of the additive calculus

$$\dfrac{}{\Gamma, x : A \vdash A \Rightarrow^{+} x;\ x : A}\ \textsc{LinVar}^{+} \qquad \dfrac{}{\Gamma, x :_r A \vdash A \Rightarrow^{+} x;\ x :_1 A}\ \textsc{GrVar}^{+}$$

$$\dfrac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^{+} t;\ \Delta, y : A}{\Gamma, x :_s A \vdash B \Rightarrow^{+} [x/y]t;\ \Delta + x :_1 A}\ \textsc{der}^{+}$$

$$\dfrac{\Gamma, x : A \vdash B \Rightarrow^{+} t;\ \Delta, x : A}{\Gamma \vdash A \multimap B \Rightarrow^{+} \lambda x.t;\ \Delta}\ \multimap_{R}^{+}$$

$$\dfrac{\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1;\ \Delta_1, x_2 : B \qquad \Gamma \vdash A \Rightarrow^{+} t_2;\ \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{+} [(x_1\, t_2)/x_2]t_1;\ (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap_{L}^{+}$$

$$\dfrac{\Gamma \vdash A \Rightarrow^{+} t;\ \Delta}{\Gamma \vdash \Box_r A \Rightarrow^{+} [t];\ r \cdot \Delta}\ \Box_{R}^{+}$$

$$\dfrac{\Gamma, x_2 :_r A \vdash B \Rightarrow^{+} t;\ \Delta \qquad \textit{if}\ x_2 :_s A \in \Delta\ \textit{then}\ s \sqsubseteq r\ \textit{else}\ 0 \sqsubseteq r}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^{+} \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t;\ (\Delta \backslash x_2), x_1 : \Box_r A}\ \Box_{L}^{+}$$

$$\dfrac{\Gamma \vdash A \Rightarrow^{+} t_1;\ \Delta_1 \qquad \Gamma \vdash B \Rightarrow^{+} t_2;\ \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{+} (t_1, t_2);\ \Delta_1 + \Delta_2}\ \otimes_{R}^{+}$$

$$\dfrac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^{+} t_2;\ \Delta, x_1 : A, x_2 : B}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^{+} \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2;\ \Delta, x_3 : A \otimes B}\ \otimes_{L}^{+}$$

$$\dfrac{\Gamma \vdash A \Rightarrow^{+} t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{+} \mathbf{inl}\,t;\ \Delta}\ \oplus 1_{R}^{+} \qquad \dfrac{\Gamma \vdash B \Rightarrow^{+} t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{+} \mathbf{inr}\,t;\ \Delta}\ \oplus 2_{R}^{+}$$

$$\dfrac{\Gamma, x_2 : A \vdash C \Rightarrow^{+} t_1;\ \Delta_1, x_2 : A \qquad \Gamma, x_3 : B \vdash C \Rightarrow^{+} t_2;\ \Delta_2, x_3 : B}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^{-} \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcup \Delta_2, x_1 : A \oplus B}\ \oplus_{L}^{+}$$

$$\dfrac{}{\Gamma \vdash 1 \Rightarrow^{+} ();\ \emptyset}\ 1_{R}^{+} \qquad \dfrac{\Gamma \vdash C \Rightarrow^{+} t;\ \Delta}{\Gamma, x : 1 \vdash C \Rightarrow^{+} \mathbf{let}\,() = x\,\mathbf{in}\,t;\ \Delta, x : 1}\ 1_{L}^{+}$$

Alternative additive pruning rules for pair introduction and application

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1;\, \Delta_1, x_2 : B \qquad \Gamma - \Delta_1 \vdash A \Rightarrow^{+} t_2;\, \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{+} [(x_1\, t_2)/x_2]t_1;\, (\Delta_1 + \Delta_2), x_1 : A \multimap B} \multimap_L'^{+}$$

$$\frac{\Gamma \vdash A \Rightarrow^{+} t_1;\, \Delta_1 \qquad \Gamma - \Delta_1 \vdash B \Rightarrow^{+} t_2;\, \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{+} (t_1, t_2);\, \Delta_1 + \Delta_2} \otimes_R'^{+}$$

A.2 Focusing forms of the linear-base synthesis calculi

A.2.1 Collected rules of the subtractive focusing calculus

RightAsync

<<no parses (char 27): G ; O, x : A |- B async =>-*** t ; D >>

<<no parses (char 28): G ; O |- {A -o B} async =>-*** \ x . t ;

LeftAsync

<<no parses (char 38): G ; {O, x1 : A

$x_1 \notin |\Delta|$

<<no parses (char 36): G ; {O, x3 : Tup A B} a

<<no parses (char 30): G ; {O, x2 : A} async |- C =>-*** t1 ; D1 >> <<no pars

---

$\Gamma; \Omega, x_1 : A \oplus B \Uparrow \vdash C \Rightarrow^- \textbf{case}\ x_1$ **o**

<<no parses (char 34): G ; {O, x2 : [A]

$0 \sqsubseteq$

<<no parses (char 88): G; {O, x1 : [] r A}

<<no parses (char 11): (

<<no parses (char 19): G ; x : Uni

<<no parses (char 39): G ; {x : [A] s, y :

$y \notin |\Delta|$ $\exists$

<<no parses (char 31): G ; {x : [A] r} asy

<<no parses (char 27): G, x :

A not le

<<no parses (char 31): G ; {O, x

Focus

<<no parses (char 17): G ; . |- C sync =***>- t ; D >>

C not atomic

--- FOCUS^-_R

<<no parses (char 21): G ; . async |- C =>-*** t ; D >>

RightSync

A.2.2 Collected rules of the additive focusing calculus

RightAsync

```
<<no parses (char 27):  G ; O, x :  A |- B async =>+*** t ;
<<no parses (char 28):  G ; O |- {A -o B} async =>+*** \ x
```

LeftAsync

```
<<no parses (char 28):  G ; O, x1 :
<<no parses (char 26):  G ; O, x3 :  Tup A B
<<no parses (char 30):  G ; {O,
<<no parses (char 31):  G ; {O,
```

$\Gamma; \Omega, x_1 : A \oplus B \Uparrow \vdash C \Rightarrow^- \textbf{case}\ x$

```
<<no parses (char 34):  G ; {
```

$\textit{if}\ x_2 :_s A$

```
<<no parses (char 25):  G ; O, x1 :  [] r A
<<no parses (char 37):  G ; {x :  [A]s, y :  A} async |- B =>+*** t ; D, y :
<<no parses (char 28):  G ; x :  [A]s async |- B =>+*** [x / y] t ; D + x :  [A]
<<no parses (char 27):  G
<<no parses (char 31):  G
```

Focus

```
<<no parses (char 17):  G ; .  |- C sync =***>+ t ; D >>
```

C not atomic

FO

```
<<no parses (char 21):  G ; .  async |- C =>+*** t ; D >>
```

RightSync

```
<<no parses (char 17):
<<no parses (char 17):
<<no parses (char 31):  G ; .  |-
<<no parses (char 17):  G ; .  |- A sync =***>+ t ; D >>
<<no parses (char 26):  G ; .  |- {Sum A B} sync =***>+ inl t ;
<<no parses (char 19):  G; .  |- A async =>+*** t ;
```

Alternative additive focusing pruning rules for pair introduction and application

```
                <<no parses (char 17):  G ; x2 :  B |- C =***>+ t1 ; D1, x2 :  B
                    <<no parses (char 17):  G - D1 ; .  |- A =***>+ t2 ; D2 >>
--------------------------------------------------------------------------------
<<no parses (char 22):  G ; x1 :  A -o B |- C =***>+ [(x1 t2) / x2] t1 ; (D1 + D

                       <<no parses (char 12):  G ; .  |- A =***>+ t1 ; D1 >>
                    <<no parses (char 17):  G - D1 ; .  |- B =***>+ t2 ; D2 >>
             ------------------------------------------------------------------
             <<no parses (char 18):  G ; .  |- Tup A B =***>+ pair t1 t2 ; D1 +
```

Appendix B

Benchmark problems

B.1 List of linear-base synthesis benchmark problems

B.2 Linear-base Proofs

This section gives the proofs of Lemma 3.3.1 and Lemma 3.3.2, along with soundness results for the variant systems: additive pruning and subtractive division.

We first state and prove some intermediate results about context manipulations which are needed for the main lemmas.

Definition B.2.1 (Context approximation)**.** For contexts Γ_1, Γ_2 then:

$$\frac{}{\emptyset \sqsubseteq \emptyset} \qquad \frac{\Gamma_1 \sqsubseteq \Gamma_2}{\Gamma_1, x : A \sqsubseteq \Gamma_2, x : A}$$

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \qquad r \sqsubseteq s}{\Gamma_1, x :_r A \sqsubseteq \Gamma_2, x :_s A} \qquad \frac{\Gamma_1 \sqsubseteq \Gamma_2 \qquad 0 \sqsubseteq s}{\Gamma_1 \sqsubseteq \Gamma_2, x :_s A}$$

This is actioned in type checking by iterative application of Approx.

Lemma B.2.1 **$(\Gamma + (\Gamma' - \Gamma'') \sqsubseteq (\Gamma + \Gamma') - \Gamma'')$.**

Proof. Induction over the structure of both Γ' and Γ''. The possible forms of Γ' and Γ'' are considered in turn:

1. $\Gamma' = \emptyset$ and $\Gamma'' = \emptyset$

   We have:

$$(\Gamma + \emptyset) - \emptyset = \Gamma + (\emptyset - \emptyset)$$

   From definitions 5.1.2 and 3.3.1, we know that on the left hand side:

$$\begin{aligned}(\Gamma + \emptyset) - \emptyset &= \Gamma + \emptyset \\ &= \Gamma\end{aligned}$$

   and on the right-hand side:

$$\begin{aligned}\Gamma + (\emptyset - \emptyset) &= \Gamma + \emptyset \\ &= \Gamma\end{aligned}$$

   making both the left and right hand sides equivalent:

$$\Gamma = \Gamma$$

2. $\Gamma' = \Gamma', x : A$ and $\Gamma'' = \emptyset$

   We have

$$(\Gamma + \Gamma', x : A) - \emptyset = \Gamma + (\Gamma, x : A - \emptyset)$$

From definitions 5.1.2 and 3.3.1, we know that on the left hand side we have:

$$\begin{aligned}(\Gamma + \Gamma', x : A) - \emptyset &= (\Gamma, \Gamma'), x : A - \emptyset \\ &= (\Gamma, \Gamma'), x : A\end{aligned}$$

and on the right hand side:

$$\begin{aligned}\Gamma + (\Gamma, x : A - \emptyset) &= \Gamma + \Gamma', x : A \\ &= (\Gamma, \Gamma', x : A)\end{aligned}$$

making both the left and right hand sides equal:

$$(\Gamma, \Gamma'), x : A = (\Gamma, \Gamma'), x : A$$

3. $\Gamma' = \Gamma', x : A$ and $\Gamma'' = \Gamma'', x : A$

   We have

$$(\Gamma + \Gamma', x : A) - \Gamma'', x : A = \Gamma + (\Gamma', x : A - \Gamma'', x : A)$$

From definitions 5.1.2 and 3.3.1, we know that on the left hand side we have:

$$\begin{aligned}(\Gamma + \Gamma', x : A) - \Gamma'', x : A &= (\Gamma, \Gamma'), x : A - \Gamma'', x : A \\ &= \Gamma, \Gamma' - \Gamma''\end{aligned}$$

and on the right hand side:

$$\begin{aligned}\Gamma + (\Gamma', x : A - \Gamma'', x : A) &= \Gamma + (\Gamma' - \Gamma'') \\ &= \Gamma, \Gamma' - \Gamma''\end{aligned}$$

making both the left and right hand sides equivalent:

$$\Gamma, \Gamma' - \Gamma'' = \Gamma, \Gamma' - \Gamma''$$

4. $\Gamma' = \Gamma', x :_r A$ and $\Gamma'' = \emptyset$

   We have

$$(\Gamma + \Gamma', x :_r A) - \emptyset = \Gamma + (x :_r A - \emptyset)$$

From definitions 5.1.2 and 3.3.1, we know that on the left hand side we have:

$$\begin{aligned}(\Gamma + \Gamma', x :_r A) - \emptyset &= (\Gamma + \Gamma', x :_r A)\\ &= (\Gamma, \Gamma'), x :_r A\end{aligned}$$

and on the right hand side:

$$\Gamma + (\Gamma', x :_r A - \emptyset) = \Gamma + (\Gamma', x :_r A) \qquad = (\Gamma, \Gamma'), x :_r A$$

making both the left and right hand sides equivalent:

$$(\Gamma, \Gamma'), x :_r A = (\Gamma, \Gamma'), x :_r A$$

5. $\Gamma' = \Gamma', x :_r A$ and $\Gamma'' = \Gamma'', x :_s A$

Thus we have (for the LHS of the inequality term):

$$\Gamma + (\Gamma', x :_r A - \Gamma'', x :_s A)$$

which by context subtraction yields:

$$\Gamma + (\Gamma', x :_r A - \Gamma'', x :_s A) = \Gamma + (\Gamma' - \Gamma''), x :_{q'} A$$

where:

$$\exists q'.r \sqsupseteq q' + s \quad \forall \hat{q}'.r \sqsupseteq \hat{q}' + s \implies q' \sqsupseteq \hat{q}' \qquad (2)$$

And for the LHS of the inequality, from definitions 5.1.2 and 3.3.1 we have:

$$\begin{aligned}(\Gamma + \Gamma', x :_r A) - \Gamma'', x :_s A &= (\Gamma + \Gamma'), x :_r A - \Gamma'', x :_s A \\ &= ((\Gamma + \Gamma') - \Gamma''), x :_r A - x :_s A \\ &= ((\Gamma + \Gamma') - \Gamma''), x :_q A\end{aligned}$$

where:

$$\exists q.r \sqsupseteq q + s \quad \forall \hat{q}.r \sqsupseteq \hat{q} + s \implies q \sqsupseteq \hat{q} \qquad (1)$$

Applying $\exists q.r \sqsupseteq q + s$ to maximality (2) (at $\hat{q}' = q$) then yields that $q \sqsubseteq q'$. Therefore, applying induction, we derive:

$$\frac{(\Gamma + (\Gamma' - \Gamma'')) \sqsubseteq ((\Gamma + \Gamma') - \Gamma'') \qquad q \sqsubseteq q'}{(\Gamma + (\Gamma' - \Gamma'')), x :_q A \sqsubseteq ((\Gamma + \Gamma') - \Gamma''), x :_{q'} A}$$

satisfying the lemma statement.

□

Lemma B.2.2 $((\Gamma - \Gamma') + \Gamma' \sqsubseteq \Gamma)$**.**

Proof. The proof follows by induction over the structure of Γ'. The possible forms of Γ' are considered in turn:

1. $\Gamma' = \emptyset$

   We have:

$$(\Gamma - \emptyset) + \emptyset = \Gamma$$

   From definition 3.3.1, we know that:

$$\Gamma - \emptyset = \Gamma$$

   and from definition 5.1.2, we know:

$$\Gamma + \emptyset = \Gamma$$

   giving us:

$$\Gamma = \Gamma$$

2. $\Gamma' = \Gamma'', x : A$

   and let $\Gamma = \Gamma', x : A$.

$$(\Gamma', x : A - \Gamma'', x : A) + \Gamma'', x : A = \Gamma$$

   From definition 5.1.2, we know that:

$$\begin{aligned}(\Gamma', x : A - \Gamma'', x : A) + \Gamma'', x : A &= ((\Gamma' - \Gamma'') + \Gamma''), x : A \\ \textit{induction} &= \Gamma', x : A \\ &= \Gamma\end{aligned}$$

   thus satisfying the lemma statement by equality.

3. $\Gamma' = \Gamma'', x :_r A$

   and let $\Gamma = \Gamma', x :_s A$.

   We have:

$$(\Gamma', x :_s A - \Gamma'', x :_r A) + \Gamma'', x :_r A$$

   From definition 3.3.1, we know that:

$$\begin{aligned} &(\Gamma', x :_s A - \Gamma'', x :_r A) + \Gamma'', x :_r A \\ =\ &(\Gamma' - \Gamma''), x :_q A + \Gamma'', x :_r A \\ =\ &((\Gamma' - \Gamma'') + \Gamma''), x :_{q+r} A \end{aligned}$$

   where $s \sqsupseteq q + r$ and $\forall q'.s \sqsupseteq q' + r \implies q \sqsupseteq q'$.

   Then by induction we derive the ordering:

$$\frac{((\Gamma' - \Gamma'') + \Gamma'') \sqsubseteq \Gamma' \qquad q + r \sqsubseteq s}{((\Gamma' - \Gamma'') + \Gamma''), x :_{q+r} A \sqsubseteq \Gamma', x :_s A}$$

   which satifies the lemma statement.

□

Lemma B.2.3 (Context negation)**.** *For all contexts* Γ*:*

$$\emptyset \sqsubseteq \Gamma - \Gamma$$

Proof. By induction on the structure of Γ:

- $\Gamma = \emptyset$ Trivial.
- $\Gamma = \Gamma', x : A$ then $(\Gamma', x : A) - (\Gamma', x : A) = \Gamma' - \Gamma'$ so proceed by induction.

- $\Gamma = \Gamma', x :_r A$ then $\exists q.\ (\Gamma', x :_r A) - (\Gamma', x :_r A) = (\Gamma - \Gamma'), x :_q A$

  such that $r \sqsupseteq q + r$ and $\forall q'.r \sqsupseteq q' + r \implies q \sqsupseteq q'$.

  Instantiating maximality with $q' = 0$ and reflexivity then we have $0 \sqsubseteq q$. From this, and the inductive hypothesis, we can construct:

$$\frac{\emptyset \sqsubseteq (\Gamma - \Gamma') \quad 0 \sqsubseteq q}{\emptyset \sqsubseteq (\Gamma - \Gamma'), x :_q A}$$

□

Lemma B.2.4. *For all contexts* Γ_1, Γ_2, *where* $[\Gamma_2]$ *(i.e.,* Γ_2 *is all graded) then:*

$$\Gamma_2 \sqsubseteq \Gamma_1 - (\Gamma_1 - \Gamma_2)$$

Proof. By induction on the structure of Γ_2.

- $\Gamma_2 =\sqsubseteq$

  Then $\Gamma_1 - (\Gamma_1 - \emptyset) = \Gamma_1 - \Gamma_1$.

  By Lemma B.2.3, then $\emptyset \sqsubseteq (\Gamma_1 - \Gamma_1)$ satisfying this case.

- $\Gamma_2 = \Gamma_2', x :_s A$

  By the premises $\Gamma_1 \sqsubseteq \Gamma_2$ then we can assume $x \in \Gamma_1$ and thus (by context rearrangement) $\Gamma_1', x :_r A$.

  Thus we consider $(\Gamma_1', x :_r A) - ((\Gamma_1', x :_r A) - (\Gamma_2', x :_s A))$.

$$\begin{aligned}
&(\Gamma_1', x :_r A) - ((\Gamma_1', x :_r A) - (\Gamma_2', x :_s A)) \\
&= (\Gamma_1', x :_r A) - ((\Gamma_1' - \Gamma_2'), x :_q A) \\
&= (\Gamma_1' - (\Gamma_1' - \Gamma_2')), x :_{q'} A
\end{aligned}$$

where (1) $\exists q.\, r \sqsupseteq q + s$ with (2) $(\forall \hat{q}. r \sqsupseteq \hat{q} + s \implies q \sqsupseteq \hat{q})$

and (3) $\exists q'.\, r \sqsupseteq q' + q$ with (4) $(\forall \hat{q'}. r \sqsupseteq \hat{q'} + s \implies q' \sqsupseteq \hat{q'})$.

Apply (1) to (4) by letting $\hat{q'} = s$ and by commutativity of $+$ then we get that $q' \sqsupseteq s$.

By induction we have that

$$\Gamma'_1 \sqsubseteq \Gamma'_1 - (\Gamma'_1 - \Gamma'_2) \tag{ih}$$

Thus we get that:

$$\frac{s \sqsubseteq q' \quad \Gamma'_1 \sqsubseteq \Gamma'_1 - (\Gamma'_1 - \Gamma'_2)}{\Gamma'_1, x :_s A \sqsubseteq (\Gamma'_1 - (\Gamma'_1 - \Gamma'_2)), x :_{q'} A}$$

- $\Gamma_2 = \Gamma'_2, x : A$ Trivial as it violates the grading condition of the premise.

$\square$

Lemma 3.3.1 (Subtractive synthesis soundness)**.** *For all* Γ *and* A *then:*

$$\Gamma \vdash A \Rightarrow^- t \mid \Delta \quad \implies \quad \Gamma - \Delta \vdash t : A$$

i.e. t *has type* A *under context* $\Gamma - \Delta$*, that contains just those linear and graded variables with grades reflecting their use in* t*. Appendix B.2 provides the proof.*

Proof. Structural induction over the synthesis rules. Each of the possible synthesis rules are considered in turn.

1. Case $\textsc{LinVar}^-$

   In the case of linear variable synthesis, we have the derivation:

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^- x \mid \Gamma}\ \textsc{LinVar}^-$$

By the definition of context subtraction, $(\Gamma, x : A) - \Gamma = x : A$, thus we can construct the following typing derivation, matching the conclusion:

$$\dfrac{}{x : A \vdash x : A}\ \textsc{Var}$$

2. Case $\textsc{GrVar}^{-}$

Matching the form of the lemma, we have the derivation:

$$\dfrac{\exists s.\, r \sqsubseteq s + 1}{\Gamma, x :_r A \vdash A \Rightarrow^{-} x \mid \Gamma, x :_s A}\ \textsc{GrVar}^{-}$$

By the definition of context subtraction, $(\Gamma, x :_r A) - (\Gamma, x :_s A) = x :_q A$ where (1) $\exists q.\, r \sqsupseteq q + s$ and $\forall q'.r \sqsupseteq q' + s \implies q \sqsupseteq q'$.

Applying maximality (1) with $q = 1$ then we have that $1 \sqsubseteq q$ (*)

Thus, from this we can construct the typing derivation, matching the conclusion:

$$\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}\ \textsc{Var}}{x :_1 A \vdash x : A}\ \textsc{Der} \qquad 1 \sqsubseteq q\ (*)}{x :_q A \vdash x : A}\ \textsc{Approx}$$

3. Case $\multimap_R^{-}$

We thus have the derivation:

$$\dfrac{\Gamma, x : A \vdash B \Rightarrow^{-} t \mid \Delta \qquad x \notin |\Delta|}{\Gamma \vdash A \multimap B \Rightarrow^{-} \lambda x.t \mid \Delta}\ \multimap_R^{-}$$

By induction we then have that:

$$(\Gamma, x : A) - \Delta \vdash t : B$$

Since $x \notin |\Delta|$ then by the definition of context subtraction we have that $(\Gamma, x : A) - \Delta = (\Gamma - \Delta), x : A$. From this, we can construct the following derivation, matching the conclusion:

$$\frac{(\Gamma - \Delta), x : A \vdash t : B}{\Gamma - \Delta \vdash \lambda x.t : A \multimap B}\textsc{Abs}$$

4. Case $\multimap_L^-$

   Matching the form of the lemma, the application derivation is:

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^- t_1 \mid \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1 \vdash A \Rightarrow^- t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^- [(x_1\, t_2)/x_2]t_1 \mid \Delta_2}\multimap_L^-$$

   By induction, we have that:

$$(\Gamma, x_2 : B) - \Delta_1 \vdash t_1 : C \qquad \text{(ih1)}$$

$$\Delta_1 - \Delta_2 \vdash t_2 : A \qquad \text{(ih2)}$$

   By the definition of context subtraction and since $x_2 \notin |\Delta_1|$ then (ih1) is equal to:

$$(\Gamma - \Delta_1), x_2 : B \vdash t_1 : C \qquad \text{(ih1')}$$

   We can thus construct the following typing derivation, making use of of the admissibility of linear substitution (Lemma 5.1.1):

$$\dfrac{\dfrac{(\Gamma - \Delta_1), x_2 : B \multimap C \vdash t_1 : C}{\Gamma - \Delta_1 \vdash \lambda x_2.t_1 : B \multimap C}\,\text{ABS} \qquad \dfrac{\dfrac{}{x_1 : A \multimap B \vdash x_1 : A \multimap B}\,\text{VAR} \qquad \Delta_1 - \Delta_2 \vdash t_2 : A}{(\Delta_1 - \Delta_2), x_1 : A \multimap B \vdash x_1\, t_2 : B}\,\text{APP}}{(\Gamma - \Delta_1) + (\Delta_1 - \Delta_2), x_1 : A \multimap B \vdash [(x_1\, t_2)/x_2]t_1 : C}\,\text{APP}$$

From Lemma B.2.1, we have that

$$((\Gamma - \Delta_1) + (\Delta_1 - \Delta_2)), x_1 : A \multimap B \sqsubseteq (((\Gamma - \Delta_1) + \Delta_1) - \Delta_2), x_1 : A \multimap B$$

and from Lemma B.2.2, that:

$$(((\Gamma - \Delta_1) + \Delta_1) - \Delta_2), x_1 : A \multimap B \sqsubseteq (\Gamma - \Delta_2), x_1 : A \multimap B$$

which, since x_1 is not in Δ_2 (as x_1 is not in Γ) $(\Gamma - \Delta_2), x_1 : A \multimap B = (\Gamma, x_1 : A \multimap B) - \Delta_2$. Applying these inequalities with Approx then yields the lemma's conclusion $(\Gamma, x_1 : A \multimap B) - \Delta_2 \vdash [(x_1\, t_2)/x_2]t_1 : C$.

5. Case $\Box_R^-$

   The synthesis rule for boxing can be constructed as:

$$\dfrac{\Gamma \vdash A \Rightarrow^- t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^- [t] \mid \Gamma - r \cdot (\Gamma - \Delta)}\,\Box_R^-$$

   By induction on the premise we get:

$$\Gamma - \Delta \vdash t : A$$

   Since we apply scalar multipication ih the conclusion of the rule to $\Gamma - \Delta$ then we know that all of $\Gamma - \Delta$ must be graded assumptions.

From this, we can construct the typing derivation:

$$\dfrac{[\Gamma - \Delta] \vdash t : A}{r \cdot [\Gamma - \Delta] \vdash [t] : \Box_r A}\ \textsc{Pr}$$

Via Lemma B.2.4, we then have that $(r \cdot \Gamma - \Delta) \sqsubseteq (\Gamma - (\Gamma - (r \cdot (\Gamma - \Delta))))$ thus, we can derived:

$$\dfrac{\dfrac{[\Gamma - \Delta] \vdash t : A}{r \cdot [\Gamma - \Delta] \vdash [t] : \Box_r A}\ \textsc{Pr} \quad \text{Lem. B.2.4}}{\Gamma - (\Gamma - (r \cdot (\Gamma - \Delta))) \vdash [t] : \Box_r A}\ \textsc{Approx}$$

Satisfying the goal of the lemma.

6. Case $\Box_L^-$

The synthesis rule for unboxing has the form:

$$\dfrac{\Gamma, x_2 :_r A \vdash B \Rightarrow^- t \mid \Delta, x_2 :_s A \qquad 0 \sqsubseteq s}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^- \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid \Delta}\ \Box_L^-$$

By induction on the premise we have that:

$$(\Gamma, x_2 :_r A) - (\Delta, x_2 :_s A) \vdash t : B$$

By the definition of context subtraction we get that $\exists q$ and:

$$(\Gamma, x_2 :_r A) - (\Delta, x_2 :_s A) = (\Gamma - \Delta), x_2 :_q A$$

such that $r = q + s$

We also have that $0 \sqsubseteq s$.

By monotonicity with $q \sqsubseteq q$ (reflexivity) and $0 \sqsubseteq s$ then $q \sqsubseteq q + s$.

By context subtraction we have $r = q + s$ therefore $q \sqsubseteq r$ (*).

From this, we can construct the typing derivation:

$$\dfrac{\dfrac{}{x_1 : \Box_r A \vdash x_1 : \Box_r A}\textsc{Var} \qquad \dfrac{(\Gamma - \Delta), x_2 :_q A \vdash t : B \quad (*)}{(\Gamma - \Delta), x_2 :_r A \vdash t : B}\textsc{approx}}{(\Gamma - \Delta), x_1 : \Box_r A \vdash \textbf{let}\, [x_2] = x_1 \,\textbf{in}\, t : B}\textsc{Let}$$

Which matches the goal.

7. Case $\otimes_R^-$

The synthesis rule for pair introduction has the form:

$$\dfrac{\Gamma \vdash A \Rightarrow^- t_1 \mid \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow^- t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^- (t_1, t_2) \mid \Delta_2}\otimes_R^-$$

By induction we get:

$$\Gamma - \Delta_1 \vdash t_1 : A \tag{ih1}$$

$$\Delta_1 - \Delta_2 \vdash t_2 : B \tag{ih2}$$

From this, we can construct the typing derivation:

$$\dfrac{\Gamma - \Delta_1 \vdash t_1 : A \qquad \Delta_1 - \Delta_2 \vdash t_2 : B}{(\Gamma - \Delta_1) + (\Delta_1 - \Delta_2) \vdash (t_1, t_2) : A \otimes B}\textsc{Pair}$$

From Lemma B.2.1, we have that:

$$(\Gamma - \Delta_1) + (\Delta_1 - \Delta_2) \sqsubseteq ((\Gamma - \Delta_1) + \Delta_1) - \Delta_2$$

and from Lemma B.2.2, that:

$$((\Gamma - \Delta_1) + \Delta_1) - \Delta_2 \sqsubseteq \Gamma - \Delta_2$$

From which we then apply Approx to the above derivation, yielding the goal $\Gamma - \Delta_2 \vdash (t_1, t_2) : A \otimes B$.

8. Case $\otimes_L^-$

The synthesis rule for pair elimination has the form:

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^- t_2 \mid \Delta \qquad x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^- \textbf{let}\,(x_1, x_2) = x_3\,\textbf{in}\,t_2 \mid \Delta}\ \otimes_L^-$$

By induction we get:

$$(\Gamma, x_1 : A, x_2 : B) - \Delta \vdash t_2 : C$$

since $x_1 \notin |\Delta| \wedge x_2 \notin |\Delta|$ then $(\Gamma, x_1 : A, x_2 : B) - \Delta = (\Gamma - \Delta), x_1 : A, x_2 : B$.

From this, we can construct the following typing derivation, matching the conclusion:

$$\frac{\dfrac{}{x_3 : A \otimes B \vdash x_3 : A \otimes B}\ \textsc{Var} \qquad (\Gamma - \Delta), x_1 : A, x_2 : B \vdash t_2 : C}{(\Gamma - \Delta), x_3 : A \otimes B \vdash \textbf{let}\,(x_1, x_2) = x_3\,\textbf{in}\,t_2 : C}\ \textsc{Case}$$

which matches the conclusion since $(\Gamma - \Delta), x_3 : A \otimes B = (\Gamma, x_3 : A \otimes B) - \Delta$

since $x_3 \notin |\Delta|$ by its disjointness from Γ.

9. Case $\oplus 1_R^-$and $\oplus 2_R^-$

   The synthesis rules for sum introduction are straightforward. For $\oplus 1_R^-$we have the rule:

$$\frac{\Gamma \vdash A \Rightarrow^- t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^- \mathbf{inl}\, t \mid \Delta} \oplus 1_R^-$$

   By induction we have:

$$\Gamma - \Delta \vdash t : A \tag{ih1}$$

   from which we can construct the typing derivation, matching the conclusion:

$$\frac{\Gamma - \Delta \vdash t : A}{\Gamma - \Delta \vdash \mathbf{inl}\, t : A \oplus B} \oplus 1_R^-$$

   Matching the goal. And likewise for $\oplus 2_R^-$.

10. Case $\oplus_L^-$The synthesis rule for sum elimination has the form:

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow^- t_1 \mid \Delta_1 \quad \Gamma, x_3 : B \vdash C \Rightarrow^- t_2 \mid \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2|}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^- \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcap \Delta_2} \oplus_L^-$$

   By induction:

$$(\Gamma, x_2 : A) - \Delta_1 \vdash t_1 : C \tag{ih}$$

$$(\Gamma, x_3 : B) - \Delta_2 \vdash t_2 : C \tag{ih}$$

From this we can construct the typing derivation, matching the conclusion:

$$\dfrac{\dfrac{}{x_1 : A \oplus B \vdash t_1 : A \oplus B}\textsc{Var} \qquad (\Gamma - \Delta_1), x_2 : A \vdash t_2 : C \qquad (\Gamma - \Delta_2), x_3 : B \vdash t_3 : C}{(\Gamma, x_1 : A \oplus B) - (\Delta_1 \sqcap \Delta_2) \vdash \textbf{case}\ x_1\ \textbf{of inl}\ x_2 \to t_1;\ \textbf{inr}\ x_3 \to t_2 : C}\textsc{Case}$$

11. Case 1_R^-

$$\dfrac{}{\Gamma \vdash \mathbf{1} \Rightarrow^- () \mid \Gamma}1_R^-$$

By Lemma B.2.3 we have that $\emptyset \sqsubseteq \Gamma - \Gamma$ then we have:

$$\dfrac{\dfrac{}{\emptyset \vdash () : \mathbf{1}}1}{\Gamma - \Gamma \vdash () : \mathbf{1}}\textsc{Approx}$$

Matching the goal

12. Case 1_L^-

$$\dfrac{\Gamma \vdash C \Rightarrow^- t \mid \Delta}{\Gamma, x : \mathbf{1} \vdash C \Rightarrow^- \textbf{let}\ () = x\ \textbf{in}\ t \mid \Delta}1_L^-$$

By induction we have:

$$\Gamma - \Delta \vdash t : C \qquad \text{(ih)}$$

Then we make the derivation:

$$\dfrac{\dfrac{}{x : 1 \vdash x : 1}\textsc{Var} \qquad \Gamma - \Delta \vdash t : C}{(\Gamma - \Delta), x : 1 \vdash \textbf{let}\,() = x\,\textbf{in}\,t : C}\textsc{Let1}$$

where the context is equal to $(\Gamma, x : 1) - \Delta$.

13. Case $\textsc{der}^-$

$$\dfrac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^- t \mid \Delta, x :_{s'} A \qquad y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s + 1}{\Gamma, x :_r A \vdash B \Rightarrow^- [x/y]t \mid \Delta, x :_{s'} A}\textsc{der}^-$$

By induction:

$$(\Gamma, x :_s A, y : A) - (\Delta, x :_{s'} A) \vdash t : B \qquad \text{(ih)}$$

By the definition of context subtraction we have (since also $y \notin |\Delta|$)

$$\begin{aligned}&(\Gamma, x :_s A, y : A) - (\Delta, x :_{s'} A)\\ =\ &(\Gamma - \Delta), x :_q A, y : A\end{aligned}$$

where $\exists q.\, s \sqsupseteq q + s'$ (1) and $\forall \hat{q}. s \sqsupseteq \hat{q} + s' \implies q \sqsupseteq \hat{q}$ (2)

The goal context is computed by:

$$\begin{aligned}&(\Gamma, x :_r A) - (\Delta, x :_{s'} A)\\ =\ &(\Gamma - \Delta), x :_{q'} A\end{aligned}$$

where $r \sqsupseteq q' + s'$ (3) and $\forall \hat{q'}. r \sqsupseteq \hat{q'} + s' \implies q' \sqsupseteq \hat{q'}$ (4)

From the premise of DER^{-}we have $r \sqsupseteq (s+1)$.

$$\begin{aligned}
\text{congruence of } + \text{ and } (1) &\implies s+1 \sqsupseteq q+s'+1 && (5)\\
\text{transitivity with DER}^{-}\text{premise and } (5) &\implies r \sqsupseteq q+s'+1 && (6)\\
+ \text{ assoc./comm. on } (6) &\implies r \sqsupseteq q+1+s' && (7)\\
\text{apply } (8) \text{ to } (4) \text{ with } \hat{q'} = q+1 &\implies q' \sqsupseteq q+1 && (8)
\end{aligned}$$

Using this last result we derive:

$$\dfrac{\dfrac{\dfrac{(\Gamma - \Delta), x :_q A, y : A \vdash t : B}{(\Gamma - \Delta), x :_q A, y :_1 A \vdash t : B}\ \textsc{Der}}{(\Gamma - \Delta), x :_{q+1} A \vdash [x/y]t : B}\ \textsc{contraction} \qquad (8)}{(\Gamma - \Delta), x :_{q'} A \vdash [x/y]t : B}\ \textsc{approx}$$

Which matches the goal.

□

Lemma 3.3.2 (Additive synthesis soundness)**.** *For all* Γ *and* A*:*

$$\Gamma \vdash A \Rightarrow^{+} t; \Delta \quad \implies \quad \Delta \vdash t : A$$

Appendix **??** *gives the proof.*

Proof. 1. Case LinVar^{+}

In the case of linear variable synthesis, we have the derivation:

$$\dfrac{}{\Gamma, x : A \vdash A \Rightarrow^{+} x;\ x : A}\ \textsc{LinVar}^{+}$$

Therefore we can construct the following typing derivation, matching the conclusion:

$$\dfrac{}{x : A \vdash x : A}\ \textsc{var}$$

2. Case $\textsc{GrVar}^{+}$

   Matching the form of the lemma, we have the derivation:

$$\dfrac{}{\Gamma, x :_r A \vdash A \Rightarrow^{+} x;\ x :_1 A}\ \textsc{GrVar}^{+}$$

   From this we can construct the typing derivation, matching the conclusion:

$$\dfrac{\dfrac{}{x : A \vdash x : A}\ \textsc{Var}}{x :_1 A \vdash x : A}\ \textsc{Der}$$

3. Case $\multimap_R^{+}$

   We thus have the derivation:

$$\dfrac{\Gamma, x : A \vdash B \Rightarrow^{+} t;\ \Delta, x : A}{\Gamma \vdash A \multimap B \Rightarrow^{+} \lambda x.t;\ \Delta}\ \multimap_R^{+}$$

   By induction on the premise we then have:

$$\Delta, x : A \vdash t : B$$

   From this, we can construct the typing derivation, matching the conclusion:

$$\dfrac{\Delta, x : A \vdash t : B}{\Delta \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

4. Case $\multimap_L^+$

   Matching the form of the lemma, the application derivation can be constructed as:

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1;\ \Delta_1, x_2 : B \qquad \Gamma \vdash A \Rightarrow^+ t_2;\ \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^+ [(x_1\, t_2)/x_2]t_1;\ (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap_L^+$$

   By induction on the premises we then have the following typing judgments:

$$\Delta_1, x_2 : B \vdash t_1 : C$$
$$\Delta_2 \vdash t_2 : A$$

   We can thus construct the following typing derivation, making use of the admissibility of linear substitution (Lemma 5.1.1):

$$\frac{\dfrac{\dfrac{}{x_1 : A \multimap B \vdash x_1 : A \multimap B}\ \textsc{var} \qquad \Delta_2 \vdash t_2 : A}{\Delta_2, x_1 : A \multimap B \vdash x_1\, t_2 : B}\ \textsc{app} \qquad \Delta_1, x_2 : B \vdash t_1 : C}{(\Delta_1 + \Delta_2), x_1 : A \multimap B \vdash [(x_1\, t_2)/x_2]t_1 : C}\ (\text{L. } 5.1.1)$$

5. Case $\Box_R^+$

   The synthesis rule for boxing can be constructed as:

$$\frac{\Gamma \vdash A \Rightarrow^+ t;\ \Delta}{\Gamma \vdash \Box_r A \Rightarrow^+ [t];\ r \cdot \Delta}\ \Box_R^+$$

   By induction we then have:

$$\Delta \vdash t : A$$

   In the conclusion of the above derivation we know that $r \cdot \Delta$ is defined, therefore it must be that all of Δ are graded assumptions, i.e., we have that $[\Delta]$ holds. We can thus construct the following typing derivation, matching the conclusion:

$$\frac{[\Delta] \vdash t : A}{r \cdot [\Delta] \vdash [t] : \Box_r A}\ \textsc{Pr}$$

6. Case $\textsc{der}^+$

   From the dereliction rule we have:

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^+ t;\ \Delta, y : A}{\Gamma, x :_s A \vdash B \Rightarrow^+ [x/y]t;\ \Delta + x :_1 A}\ \textsc{der}^+$$

   By induction we get:

$$\Delta, y : A \vdash t : B \qquad \text{(ih)}$$

   Case on $x \in \Delta$

   - $x \in \Delta$, i.e., $\Delta = \Delta', x :_{s'} A$.

     Then by admissibility of contraction we can derive:

$$\frac{\dfrac{\Delta', x :_{s'} A, y : A \vdash t : B}{\Delta', x :_{s'} A, y :_1 A \vdash t : B}\ \textsc{Der}}{(\Delta', x :_{s'} A) + x :_1 A \vdash [x/y]t : B}$$

     Satisfying the lemma statment.

   - $x \notin \Delta$. Then again from the admissiblity of contraction, we derive the typing:

$$\dfrac{\dfrac{\Delta, y : A \vdash t : B}{\Delta, y :_1 A \vdash t : B}\ \textsc{Der}}{\Delta + x :_1 A \vdash [x/y]t : B}$$

which is well defined as $x \notin \Delta$ and gives the lemma conclusion.

7. Case $\Box_L^+$

   The synthesis rule for unboxing has the form:

$$\dfrac{\Gamma, x_2 :_r A \vdash B \Rightarrow^+ t; \Delta \qquad \textit{if } x_2 :_s A \in \Delta \textit{ then } s \sqsubseteq r \textit{ else } 0 \sqsubseteq r}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^+ \textbf{let}\,[x_2] = x_1\,\textbf{in}\,t;\ (\Delta \backslash x_2), x_1 : \Box_r A}\ \Box_L^+$$

   By induction we have that:

$$\Delta \vdash t : B \qquad \text{(ih)}$$

   Case on $x_2 :_s A \in \Delta$

   - $x_2 :_s A \in \Delta$, i.e., $s \sqsubseteq r$.

     From this, we can construct the typing derivation, matching the conclusion:

$$\dfrac{\dfrac{}{x_1 : \Box_r A \vdash x_1 : \Box_r A}\ \textsc{var} \qquad \Delta, x_2 :_r A \vdash t : B}{\Delta, x_1 : \Box_r A \vdash \textbf{let}\,[x_2] = x_1\,\textbf{in}\,t : B}\ \textsc{let}\Box$$

   - $x_2 :_s A \notin \Delta$, i.e., $0 \sqsubseteq r$.

From this, we can construct the typing derivation, matching the conclusion:

$$\dfrac{\dfrac{}{x_1 : \Box_r A \vdash x_1 : \Box_r A}\ \textsc{var} \qquad \dfrac{\dfrac{\Delta \vdash t : B}{\Delta, x_2 :_0 A \vdash t : B}\ \textsc{Weak} \qquad 0 \sqsubseteq r}{\Delta, x_2 :_r A \vdash t : B}\ \textsc{Approx}}{\Delta, x_1 : \Box_r A \vdash \textbf{let}\, [x_2] = x_1\, \textbf{in}\, t : B}\ \textsc{let}\Box$$

8. Case $\otimes_R^+$

The synthesis rule for pair introduction has the form:

$$\dfrac{\Gamma \vdash A \Rightarrow^+ t_1;\, \Delta_1 \qquad \Gamma \vdash B \Rightarrow^+ t_2;\, \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^+ (t_1, t_2);\, \Delta_1 + \Delta_2}\ \otimes_R^+$$

By induction on the premises we have that:

$$\Delta_1 \vdash t_1 : A \qquad \text{(ih1)}$$

$$\Delta_2 \vdash t_2 : B \qquad \text{(ih2)}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\dfrac{\Delta_1 \vdash t_1 : A \qquad \Delta_2 \vdash t_2 : B}{\Delta_1 + \Delta_2 \vdash (t_1, t_2) : A \otimes B}\ \textsc{pair}$$

9. Case $\otimes_L^+$

The synthesis rule for pair elimination has the form:

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^{+} t_2;\ \Delta, x_1 : A, x_2 : B}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^{+} \textbf{let}\,(x_1, x_2) = x_3\,\textbf{in}\,t_2;\ \Delta, x_3 : A \otimes B}\ \otimes_L^{+}$$

By induction on the premises we have that:

$$\Delta_1 \vdash t_1 : A \tag{ih1}$$

$$\Delta_2 \vdash t_2 : B \tag{ih2}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\dfrac{}{x_3 : A \otimes B \vdash x_3 : A \otimes B}\ \textsc{Var} \qquad \Delta, x_1 : A, x_2 : B \vdash t_2 : C}{\Delta, x_3 : A \otimes B \vdash \textbf{let}\,(x_1, x_2) = x_3\,\textbf{in}\,t_2 : C}\ \textsc{LetPair}$$

10. Case $\oplus 1_R^{+}$and $\oplus 2_R^{+}$

The synthesis rules for sum introduction are straightforward. For $\oplus 1_R^{+}$we have the rule:

$$\frac{\Gamma \vdash A \Rightarrow^{+} t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{+} \textbf{inl}\,t;\ \Delta}\ \oplus 1_R^{+}$$

By induction on the premises we have that:

$$\Delta \vdash t : A \tag{ih}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\Delta \vdash t : A}{\Delta \vdash \textbf{inl}\,t : A \oplus B}\ \textsc{Inl}$$

Likewise, for the $\oplus 2^{+}_{R}$we have the synthesis rule:

$$\frac{\Gamma \vdash B \Rightarrow^{+} t;\, \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{+} \textbf{inr}\, t;\, \Delta}\ \oplus 2^{+}_{R}$$

By induction on the premises we have that:

$$\Delta \vdash t : B \tag{ih}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\Delta \vdash t : B}{\Delta \vdash \textbf{inl}\, t : A \oplus B}\ \textsc{Inr}$$

11. Case $\oplus^{+}_{L}$

The synthesis rule for sum elimination has the form:

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow^{+} t_1;\, \Delta_1, x_2 : A \qquad \Gamma, x_3 : B \vdash C \Rightarrow^{+} t_2;\, \Delta_2, x_3 : B}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^{-} \textbf{case}\ x_1\ \textbf{of inl}\ x_2 \rightarrow t_1;\ \textbf{inr}\ x_3 \rightarrow t_2 \mid \Delta_1 \sqcup \Delta_2, x_1 : A \oplus B}\ \oplus^{+}_{L}$$

By induction on the premises we have that:

$$\Delta_1, x_2 : A \vdash t_1 : C \tag{ih1}$$

$$\Delta_2, x_3 : B \vdash t_2 : C \tag{ih2}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\dfrac{}{x_1 : A \oplus B \vdash x_1 : A \oplus B}\ \textsc{Var} \qquad \Delta_1, x_2 : A \vdash t_1 : C \qquad \Delta_2, x_3 : B \vdash t_2 : C}{(\Delta_1 \sqcup \Delta_2), x_1 : A \oplus B \vdash \textbf{case}\ x_1\ \textbf{of inl}\ x_2 \rightarrow t_1;\ \textbf{inr}\ x_3 \rightarrow t_2 : C}\ \textsc{Case}$$

12. Case 1^+_R

    The synthesis rule for unit introduction has the form:

$$\frac{}{\Gamma \vdash \mathbf{1} \Rightarrow^+ ();\, \emptyset}\; 1^+_R$$

    From this, we can construct the typing derivation, matching the conclusion:

$$\frac{}{\emptyset \vdash () : \mathbf{1}}\; 1$$

13. Case 1^+_L

    The synthesis rule for unit elimination has the form:

$$\frac{\Gamma \vdash C \Rightarrow^+ t;\, \Delta}{\Gamma, x : \mathbf{1} \vdash C \Rightarrow^+ \mathbf{let}\, () = x \,\mathbf{in}\, t;\, \Delta, x : \mathbf{1}}\; 1^+_L$$

    By induction on the premises we have that:

$$\Delta \vdash t : C \tag{ih}$$

    From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\dfrac{}{x : \mathbf{1} \vdash x : \mathbf{1}}\;\textsc{Var} \qquad \Delta \vdash t : C}{\Delta, x : \mathbf{1} \vdash \mathbf{let}\, () = x \,\mathbf{in}\, t : C}\;\textsc{Let1}$$

□

Lemma 3.3.3 (Additive pruning synthesis soundness)**.** *For all* Γ *and* A*:*

$$\Gamma \vdash A \Rightarrow^{+} t;\, \Delta \quad \Longrightarrow \quad \Delta \vdash t : A$$

Appendix **??** *gives the proof.*

Proof. The cases for the rules in the additive pruning synthesis calculus are equivalent to lemma (3.3.2), except for the cases of the $\multimap'^{+}_{L}$and $\otimes'^{+}_{R}$rules which we consider here:

1. Case $\multimap'^{+}_{L}$

   Matching the form of the lemma, the application derivation can be constructed as:

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1;\, \Delta_1, x_2 : B \qquad \Gamma - \Delta_1 \vdash A \Rightarrow^{+} t_2;\, \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{+} [(x_1\, t_2)/x_2]t_1;\, (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap'^{+}_{L}$$

   By induction on the premises we then have the following typing judgments:

$$\Delta_1, x_2 : B \vdash t_1 : C$$
$$\Delta_2 \vdash t_2 : A$$

   We can thus construct the following typing derivation, making use of the admissibility of linear substitution (Lemma 5.1.1):

$$\frac{\dfrac{\dfrac{}{x_1 : A \multimap B \vdash x_1 : A \multimap B}\ \textsc{var} \qquad \Delta_2 \vdash t_2 : A}{\Delta_2, x_1 : A \multimap B \vdash x_1\, t_2 : B}\ \textsc{app} \qquad \Delta_1, x_2 : B \vdash t_1 : C}{(\Delta_1 + \Delta_2), x_1 : A \multimap B \vdash [(x_1\, t_2)/x_2]t_1 : C}\ (\text{L. } 5.1.1)$$

2. Case $\otimes'^{+}_{R}$

The synthesis rule for the pruning alternative for pair introduction has the form:

$$\frac{\Gamma \vdash A \Rightarrow^{+} t_1;\, \Delta_1 \qquad \Gamma - \Delta_1 \vdash B \Rightarrow^{+} t_2;\, \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{+} (t_1, t_2);\, \Delta_1 + \Delta_2}\ \otimes_R'^{+}$$

By induction on the premises we have that:

$$\Delta_1 \vdash t_1 : A \qquad \text{(ih1)}$$

$$\Delta_2 \vdash t_2 : B \qquad \text{(ih2)}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\Delta_1 \vdash t_1 : A \qquad \Delta_2 \vdash t_2 : B}{\Delta_1 + \Delta_2 \vdash (t_1, t_2) : A \otimes B}\ \textsc{pair}$$

$\square$

Lemma B.2.5 (Soundness of focusing for subtractive synthesis)**.** *For all contexts* Γ, Ω *and types* A *then:*

1. *Right Async* : `<<no parses (char 21): G ; O |- A async =>-*** t ; D >>`
2. *Left Async* : `<<no parses (char 21): G ; O async |- C =>-*** t ; D >>`
3. *Right Sync* : `<<no parses (char 18): G ; . |- A sync =***>- t ; D >>`
4. *Left Sync* : `<<no parses (char 26): G ; {x : A }sync |- C =>-*** t ; D >>`
5. *Focus Right* : `<<no parses (char 21): G ; O async |- C =>-*** t ; D >>`
6. *Focus Left* : `<<no parses (char 28): G, x : A ; O async |- C =>-*** t ; D >`

Proof. 1. Case 1. Right Async:

(a) Case $\multimap_R^-$

In the case of the right asynchronous rule for abstraction introduction, the synthesis rule has the form:

```
<<no parses (char 27):  G ; O, x :  A |- B async =>-*** t ; D >>   x ∉ |Δ
──────────────────────────────────────────────────────────────────────────
 <<no parses (char 28):  G ; O |- {A -o B} async =>-*** \ x .  t ; D >>
```

By induction on the first premise, we have that:

$$(\Gamma, \Omega), x : A \vdash A \Rightarrow^- t \mid \Delta \qquad \text{(ih)}$$

from case 1 of the lemma. From which, we can construct the following instantiation of the $\multimap_R^-$synthesis rule in the non-focusing calculus:

$$\frac{(\Gamma, \Omega), x : A \vdash B \Rightarrow^- t \mid \Delta \qquad x \notin |\Delta|}{\Gamma, \Omega \vdash A \multimap B \Rightarrow^- \lambda x.t \mid \Delta}\multimap_R^-$$

(b) Case $\Uparrow_R^-$

In the case of the right asynchronous rule for transition to a left asynchronous judgement, the synthesis rule has the form:

```
<<no parses (char 21):  G ; O async |- C =>-*** t ; D >>
                    C not right async
──────────────────────────────────────────────────────── ⇑⁻_R
<<no parses (char 22):  G ; O |- C async =>-*** t ; D >>
```

By induction on the first premise, we have that:

$$\Gamma, \Omega \vdash C \Rightarrow^{-} t \mid \Delta$$

from case 2 of the lemma.

2. Case 2. Left Async:

   (a) Case $\otimes_L^-$

   In the case of the left asynchronous rule for pair elimination, the synthesis rule has the form:

```
<<no parses (char 38):  G ; {O, x1 :  A, x2 :  B} async |- C =
```

$$x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|$$

```
<<no parses (char 36):  G ; {O, x3 :  Tup A B} async |- C =>-*** letpair
```

   By induction on the first premise, we have that:

$$(\Gamma, \Omega), x_1 : A, x_2 : B \vdash C \Rightarrow^{-} t \mid \Delta \tag{ih}$$

   from From which, we can construct the following instantiation of the $\otimes_R^-$ synthesis rule in the non-focusing calculus:

$$\frac{(\Gamma, \Omega), x_1 : A, x_2 : B \vdash C \Rightarrow^{-} t \mid \Delta \qquad x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|}{\Gamma, (\Omega, x_3 : A \otimes B) \vdash C \Rightarrow^{-} \textbf{let}\,(x_1, x_2) = x_3\,\textbf{in}\,t \mid \Delta_2}\otimes_L^-$$

   (b) Case $\oplus_L^-$

In the case of the left asynchronous rule for sum elimination, the synthesis rule has the form:

$$\frac{\texttt{<<no parses (char 30): G ; \{O, x2 : A\} async |- C =>-*** t1 ; D1 >>} \quad \texttt{<}}{\Gamma; \Omega, x_1 : A \oplus B \Uparrow \vdash C \Rightarrow^-}$$

By induction on the first and second premises, we have that:

$$(\Gamma, \Omega), x_2 : A \vdash C \Rightarrow^- t_1 \mid \Delta_1 \qquad \text{(ih1)}$$

$$(\Gamma, \Omega), x_3 : B \vdash C \Rightarrow^- t_2 \mid \Delta_2 \qquad \text{(ih2)}$$

from case 2 of the lemma. From which, we can construct the following instantiation of the $\oplus_L^-$ synthesis rule in the non-focusing calculus:

$$\frac{(\Gamma, \Omega), x_2 : A \vdash C \Rightarrow^- t_1 \mid \Delta_1 \quad (\Gamma, \Omega), x_3 : B \vdash C \Rightarrow^- t_2 \mid \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2|}{\Gamma, (\Omega, x_1 : A \oplus B) \vdash C \Rightarrow^- \textbf{case } x_1 \textbf{ of inl } x_2 \to t_1; \textbf{ inr } x_3 \to t_2 \Delta_1 \sqcap \Delta_2} \oplus$$

(c) Case 1_L^-

In the case of the left asynchronous rule for unit elimination, the synthesis rule has the form:

$$\frac{\texttt{<<no parses (char 11): G ; .|- C =***>- t ; D >>}}{\texttt{<<no parses (char 19): G ; x : Unit |- C =***>- let () = x in t ; D >>}}$$

By induction on the premise, we have that:

$$\Gamma \vdash C \Rightarrow^- t \mid \Delta \qquad \text{(ih)}$$

from case 2 of the lemma. From which, we can construct the following instantiation of the 1_L^- synthesis rule in the non-focusing calculus matching the conclusion:

$$\frac{\Gamma \vdash C \Rightarrow^{-} t \mid \Delta}{\Gamma, x : 1 \vdash C \Rightarrow^{-} \textbf{let}\,() = x\,\textbf{in}\,t \mid \Delta}\ 1_L^{-}$$

(d) Case $\Box_L^{-}$

In the case of the left asynchronous rule for graded modality elimination, the synthesis rule has the form:

<<no parses (char 34): G ; {0, x2 : [A] r} async |- B =>-*** t ; D,

$0 \sqsubseteq s$

<<no parses (char 88): G; {0, x1 : [] r A} async |- B =>-*** let [x2]

By induction on the first premise, we have that:

$$(\Gamma, \Omega), x_2 :_r A \vdash B \Rightarrow^{-} t \mid \Delta, x_2 :_s A \qquad \text{(ih)}$$

from case 2 of the lemma. From which, we can construct the following instatiation of the $\Box_L^{-}$synthesis rule in the non-focusing calculus:

$$\frac{(\Gamma, \Omega), x_2 :_r A \vdash B \Rightarrow^{-} t \mid \Delta, x_2 :_s A \qquad 0 \sqsubseteq s}{\Gamma, (\Omega, x_1 : \Box_r A) \vdash B \Rightarrow^{-} \textbf{let}\,[x_2] = x_1\,\textbf{in}\,t \mid \Delta}\ \Box_L^{-}$$

(e) Case DER^{-}

In the case of the left asynchronous rule for dereliction, the synthesis rule has the form:

<<no parses (char 39): G ; {x : [A] s, y : A } async |- B =>-*** t ;

$y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s + 1$

<<no parses (char 31): G ; {x : [A] r} async |- B =>-*** [x/y] t ; D

By induction on the first premise, we have that:

$$\Gamma, x :_s A, y : A \vdash B \Rightarrow^- t \mid \Delta, x :_{s'} A \tag{ih}$$

from case 2 of the lemma. From which, we can construct the following instatiation of the DER$^-$synthesis rule in the non-focusing calculus:

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^- t \mid \Delta, x :_{s'} A \qquad y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s + 1}{\Gamma, x :_r A \vdash B \Rightarrow^- [x/y]t \mid \Delta, x :_{s'} A}\ \textsc{der}^-$$

(f) Case $\Uparrow_L^-$

In the case of the left asynchronous rule for transitioning an assumption from the focusing context Ω to the non-focusing context Γ, the synthesis rule has the form:

```
<<no parses (char 27):  G, x :  A ; O async |- C =>-*** t ; D >>
```

A not left async

```
<<no parses (char 31):  G ; {O, x :  A} async |- C =>-*** t ; D >>
```

$\Uparrow_L^-$

By induction on the first premise, we have that:

$$\Gamma, x : A, \Omega \vdash C \Rightarrow^- t \mid \Delta \tag{ih}$$

from case 2 of the lemma.

3. Case 3. Right Sync:

(a) Case $\otimes_R^-$

In the case of the right synchronous rule for pair introduction, the synthesis rule has the form:

$$\frac{\begin{array}{c}\texttt{<<no parses (char 17): G ; . |- A sync =***>- t1 ; D1 >>}\\ \texttt{<<no parses (char 18): D1 ; . |- B sync =***>- t2 ; D2 >>}\end{array}}{\texttt{<<no parses (char 31): G ; . |- \{Tup A B\} sync =***>- pair t1 t2 ; D2}}$$

By induction on the first and second premises, we have that:

$$\Gamma \vdash A \Rightarrow^{-} t_1 \mid \Delta_1 \qquad \text{(ih1)}$$

$$\Delta_1 \vdash B \Rightarrow^{-} t_2 \mid \Delta_2 \qquad \text{(ih2)}$$

from case 3 of the lemma. From which, we can construct the following instatiation of the $\otimes_R^-$ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma \vdash A \Rightarrow^{-} t_1 \mid \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{-} (t_1, t_2) \mid \Delta_2}\otimes_R^-$$

(b) Case $\oplus 1_R^-$ and $\oplus 2_R^-$

In the case of the right synchronous rules for sum introduction, the synthesis rules has the form:

$$\frac{\texttt{<<no parses (char 17): G ; . |- A sync =***>- t ; D >>}}{\texttt{<<no parses (char 26): G ; . |- \{Sum A B\} sync =***>- inl t ; D >>}}\oplus 1_L^+$$

$$\frac{\texttt{<<no parses (char 17): G ; . |- B sync =***>- t ; D >>}}{\texttt{<<no parses (char 26): G ; . |- \{Sum A B\} sync =***>- inr t ; D >>}}\oplus 2_L^+$$

By induction on the premises of these rules, we have that:

$$\Gamma \vdash A \Rightarrow^{-} t \mid \Delta \qquad \text{(ih1)}$$

$$\Gamma \vdash B \Rightarrow^{-} t \mid \Delta \tag{ih2}$$

from case 3 of the lemma. From which, we can construct the following instatiations of the $\oplus 1_R^-$ and $\oplus 2_R^-$ rule in the non-focusing calculus, respectively:

$$\frac{\Gamma \vdash A \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{-} \mathbf{inl}\, t \mid \Delta}\ \oplus 1_R^-$$

$$\frac{\Gamma \vdash B \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{-} \mathbf{inr}\, t \mid \Delta}\ \oplus 2_R^-$$

(c) Case 1_R^-

In the case of the right synchronous rule for unit introduction, the synthesis rule has the form:

$$\frac{}{\Gamma \vdash \mathsf{1} \Rightarrow^{-} () \mid \Gamma}\ 1_R^-$$

From which, we can construct the following instatiation of the 1_R^- synthesis rule in the non-focusing calculus:

$$\frac{}{\Gamma, \Omega \vdash \mathsf{1} \Rightarrow^{-} () \mid \Gamma}\ 1_R^-$$

(d) Case $\Box_R^-$

In the case of the right synchronous rule for graded modality introduction, the synthesis rule has the form:

```
<<no parses (char 19):  G; .  |- A async =>-*** t ; D >>
```

```
<<no parses (char 25):  G ; .  |- { [] r A} sync =***>- t ; G - r * (G -
```

By induction on the premise, we have that:

$$\Gamma \vdash A \Rightarrow^{-} t \mid \Delta \tag{ih}$$

from case 1 of the lemma. From which, we can construct the following instatiation of the $\Box_R^-$synthesis rule in the non-focusing calculus:

$$\frac{\Gamma \vdash A \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^{-} [t] \mid \Gamma - r \cdot (\Gamma - \Delta)}\ \Box_R^-$$

(e) Case $\Downarrow_R^-$

In the case of the right synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

```
<<no parses (char 22):  G ; .  |- A async =>-*** t ; D >>
----------------------------------------------------------- ⇓R-
<<no parses (char 18):  G ; .  |- A sync =***>- t ; D >>
```

By induction on the premise, we have that:

$$\Gamma \vdash A \Rightarrow^{-} t \mid \Delta \tag{ih}$$

from case 1 of the lemma.

4. Case 4. Left Sync

(a) Case $\multimap_L^-$

In the case of the left synchronous rule for application, the synthesis rule has the form:

`<<no parses (char 27): G ; {x2 : B} sync |- C =>-*** t1 ; D1 >>` $x_2 \not$

`<<no parses (char 35): G ; {x1 : A -o B} sy`

By induction on the first premise, we have that:

$$\Gamma, x_2 : B \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \tag{ih1}$$

from case 4 of the lemma. By induction on the third premise, we have that:

$$\Delta_1 \vdash A \Rightarrow^{-} t_2 \mid \Delta_2 \tag{ih2}$$

from case 3 of the lemma. From which, we can construct the following instatiation of the $\multimap^{-}_{L}$synthesis rule in the non-focusing calculus:

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1 \vdash A \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{-} [(x_1\ t_2)/x_2]t_1 \mid \Delta_2} \multimap^{-}_{L}$$

(b) Case LinVar^{-}

In the case of the left synchronous rule for linear variable synthesis, the synthesis rule has the form:

$$\frac{}{\texttt{<<no parses (char 25): G ; \{x : A\} sync |- A =>-*** x ; G >>}} \textsc{LinVar}^{-}$$

From which, we can construct the following instatiation of the LinVar^{-}

synthesis rule in the non-focusing calculus:

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^{-} x \mid \Gamma}\ \textsc{LinVar}^{-}$$

(c) Case $\textsc{GrVar}^{-}$

In the case of the left synchronous rule for graded variable synthesis, the synthesis rule has the form:

$$\exists s.\ r \sqsubseteq s + 1$$

<<no parses (char 29): G ; {x : [A] r} sync |- A =>-*** x ; G, x : [A

From which, we can construct the following instatiation of the $\textsc{GrVar}^{-}$ synthesis rule in the non-focusing calculus:

$$\frac{\exists s.\ r \sqsubseteq s + 1}{\Gamma, x :_r A \vdash A \Rightarrow^{-} x \mid \Gamma, x :_s A}\ \textsc{GrVar}^{-}$$

(d) Case $\Downarrow_L^{-}$

In the case of the left synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

<<no parses (char 26): G ; {x : A} async |- C =>-*** t ; D >>

A not atomic and not left sync

$\Downarrow_L^{-}$

<<no parses (char 25): G ; {x : A} sync |- C =>-*** t ; D >>

By induction on the premise, we have that:

$$\Gamma, x : A \vdash C \Rightarrow^{-} t \mid \Delta \qquad \text{(ih)}$$

from case 2 of the lemma.

5. Case 5. Focus Right: focus$_R^-$

   In the case of the focusing rule for transitioning from a left asynchronous judgement to a right synchronous judgement, the synthesis rule has the form:

$$\dfrac{\begin{array}{c}\texttt{<<no parses (char 17): G ; . |- C sync =***>- t ; D >>} \\ \text{C not atomic}\end{array}}{\texttt{<<no parses (char 21): G ; . async |- C =>-*** t ; D >>}}\ \textsc{focus}_R^-$$

   By induction on the first premise, we have that:

$$\Gamma \vdash C \Rightarrow^- t \mid \Delta \qquad \text{(ih)}$$

   from case 2 of the lemma.

6. Case 6. Focus Left focus$_L^-$

   In the case of the focusing rule for transitioning from a left asynchronous judgement to a left synchronous judgement, the synthesis rule has the form:

$$\dfrac{\texttt{<<no parses (char 25): G ; \{x : A\} sync |- C =>-*** t ; D >>}}{\texttt{<<no parses (char 27): G, x : A; . async |- C =>-*** t ; D >>}}\ \textsc{focus}_L^-$$

   By induction on the first premise, we have that:

$$\Gamma, x : A \vdash C \Rightarrow^- t \mid \Delta \qquad \text{(ih)}$$

   from case 2 of the lemma.

$\square$

Lemma B.2.6 (Soundness of focusing for additive synthesis)**.** *For all contexts* Γ, Ω *and types* A *then:*

1. *Right Async* : <<no parses (char 21): G ; O |- A async =>+*** t ; D >>
2. *Left Async* : <<no parses (char 21): G ; O async |- C =>+*** t ; D >>
3. *Right Sync* : <<no parses (char 18): G ; . |- A sync =***>+ t ; D >>
4. *Left Sync* : <<no parses (char 26): G ; {x : A }sync |- C =>+*** t ; D >>
5. *Focus Right* : <<no parses (char 21): G ; O async |- C =>+*** t ; D >>
6. *Focus Left* : <<no parses (char 28): G, x : A ; O async |- C =>+*** t ; D >

Proof. 1. Case 1. Right Async:

(a) Case $\multimap_R^+$

In the case of the right asynchronous rule for abstraction introduction, the synthesis rule has the form:

<<no parses (char 27): G ; O, x : A |- B async =>+*** t ; D, x : A >>

<<no parses (char 28): G ; O |- {A -o B} async =>+*** \ x . t ; D >>

By induction on the premise, we have that:

$$(\Gamma, \Omega), x : A \vdash B \Rightarrow^+ t;\ \Delta, x : A \tag{ih}$$

from case 1 of the lemma. From which, we can construct the following instatiation of the $\multimap_R^+$ synthesis rule in the non-focusing calculus:

$$\frac{(\Gamma, \Omega), x : A \vdash B \Rightarrow^+ t;\ \Delta, x : A}{\Gamma, \Omega \vdash A \multimap B \Rightarrow^+ \lambda x.t;\ \Delta}\ \mathrm{R}\multimap^+$$

(b) Case $\Uparrow_R^+$In the case of the right asynchronous rule for transition to a left asynchronous judgement, the synthesis rule has the form:

$$\dfrac{\begin{array}{c}\texttt{<<no parses (char 21): G ; O async |- C =>+*** t ; D >>}\\ C \text{ not right async}\end{array}}{\texttt{<<no parses (char 22): G ; O |- C async =>+*** t ; D >>}}\Uparrow^{+}_{R}$$

By induction on the first premise, we have that:

$$\Gamma, \Omega \vdash C \Rightarrow^{+} t; \Delta$$

from case 2 of the lemma.

2. Case 2. Left Async:

   (a) Case $\otimes^{+}_{L}$

   In the case of the left asynchronous rule for pair elimination, the synthesis rule has the form:

$$\dfrac{\texttt{<<no parses (char 28): G ; O, x1 : A, x2 : B |- C =***>+ t2 ;}}{\texttt{<<no parses (char 26): G ; O, x3 : Tup A B |- C =***>+ letpair x1 x2 =}}$$

   By induction on the premise, we have that:

$$(\Gamma, \Omega), x_1 : A, x_2 : B \vdash C \Rightarrow^{+} t_2; \Delta, x_1 : A, x_2 : B \qquad \text{(ih)}$$

   from case 2 of the lemma. From which, we can construct the following instatiation of the $\otimes^{+}_{L}$ synthesis rule in the non-focusing calculus:

$$\dfrac{(\Gamma, \Omega), x_1 : A, x_2 : B \vdash C \Rightarrow^{+} t_2; \Delta, x_1 : A, x_2 : B}{\Gamma, (\Omega, x_3 : A \otimes B) \vdash C \Rightarrow^{+} \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2; \Delta, x_3 : A \otimes B}\text{L}\otimes^{+}$$

(b) Case $\oplus_L^+$

In the case of the left asynchronous rule for sum elimination, the synthesis rule has the form:

```
<<no parses (char 30):  G ; {O, x2 :  A} async |- C =>+*** t1 ; D1, x2 :
<<no parses (char 31):  G ; {O, x3 :  B} async |- C =>+*** t2 ; D2, x3 :
```

$$\Gamma; \Omega, x_1 : A \oplus B \Uparrow \vdash C \Rightarrow^- \textbf{case } x_1 \textbf{ of inl } x_2 \to t_1;\ \textbf{inr } x_3 \to t_2 \mid \Delta_1 \sqcup \Delta_2, x_1 : A \oplus$$

By induction on the premises, we have that:

$$(\Gamma, \Omega), x_2 : A \vdash C \Rightarrow^+ t_1;\ \Delta_1, x_2 : A \qquad \text{(ih1)}$$

$$(\Gamma, \Omega), x_3 : B \vdash C \Rightarrow^+ t_2;\ \Delta_2, x_3 : B \qquad \text{(ih2)}$$

from case 2 of the lemma. From which, we can construct the following instatiation of the $\oplus_L^+$ synthesis rule in the non-focusing calculus:

$$\frac{\begin{array}{c}(\Gamma, \Omega), x_2 : A \vdash C \Rightarrow^+ t_1;\ \Delta_1, x_2 : A \\ (\Gamma, \Omega), x_3 : B \vdash C \Rightarrow^+ t_2;\ \Delta_2, x_3 : B\end{array}}{\Gamma, (\Omega, x_1 : A \oplus B) \vdash C \Rightarrow^+ \textbf{case } x_1 \textbf{ of inl } x_2 \to t_1;\ \textbf{inr } x_3 \to t_2 \mid (\Delta_1 \sqcup \Delta_2), x_1 : A \oplus}$$

(c) Case 1_L^+

In the case of the left asynchronous rule for unit elimination, the synthesis rule has the form:

```
            <<no parses (char 12):  G ; .  |- C =***>+ t ; D >>
<<no parses (char 19):  G ; x :  Unit |- C =***>+ let () = x in t ; D, x
```

By induction on the premise, we have that:

$$\Gamma \vdash C \Rightarrow^{+} t;\, \Delta \tag{ih}$$

from case 2 of the lemma. From which, we can construct the following instatiation of the 1^{+}_{L} synthesis rule in the non-focusing calculus:

$$\dfrac{\Gamma \vdash C \Rightarrow^{+} t;\, \Delta}{\Gamma, x : \mathsf{1} \vdash C \Rightarrow^{+} \mathbf{let}\, () = x\, \mathbf{in}\, t;\, \Delta, x : \mathsf{1}}\ \text{L1}^{+}$$

(d) Case $\Box^{+}_{L}$

In the case of the left asynchronous rule for graded modality elimination, the synthesis rule has the form:

```
<<no parses (char 34):  G ; {0, {x2 :  [A] r async}} |- B
```

$$\textit{if } x_2 :_s A \in \Delta \textit{ then } s \sqsubseteq r \textit{ else } 0 \sqsubseteq r$$

```
<<no parses (char 25):  G ; 0, x1 :  [] r A |- B =***>+ let [x2] = x1 in
```

By induction on the first premise, we have that:

$$(\Gamma, \Omega), x_2 :_r A \vdash B \Rightarrow^{+} t;\, \Delta \tag{ih}$$

from case 2 of the lemma. From which, we can construct the following instatiation of the $\Box^{+}_{L}$ synthesis rule in the non-focusing calculus:

$$\dfrac{(\Gamma, \Omega), x_2 :_r A \vdash B \Rightarrow^{+} t;\, \Delta \qquad \textit{if } x_2 :_s A \in \Delta \textit{ then } s \sqsubseteq r \textit{ else } 0 \sqsubseteq r}{\Gamma, (\Omega, x_1 : \Box_r A) \vdash B \Rightarrow^{+} \mathbf{let}\, [x_2] = x_1\, \mathbf{in}\, t;\, (\Delta \backslash x_2), x_1 : \Box_r A}\ \text{L}\Box^{+}$$

(e) Case DER^{+}

In the case of the left asynchronous rule for dereliction, the synthesis rule has the form:

`<<no parses (char 37): G ; {x : [A]s, y : A} async |- B =>+*** t ; I`

`<<no parses (char 28): G ; x : [A]s async |- B =>+*** [x / y] t ; D +`

By induction on the premise, we have that:

$$\Gamma, x :_s A, y : A \vdash B \Rightarrow^+ t;\ \Delta, y : A \tag{ih}$$

from case 2 of the lemma. From which, we can construct the following instantiation of the DER$^+$ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^+ t;\ \Delta, y : A}{\Gamma, x :_s A \vdash B \Rightarrow^+ [x/y]t;\ \Delta + x :_1 A}\ \textsc{der}^+$$

(f) Case $\Uparrow_L^+$

In the case of the left asynchronous rule for transitioning an assumption from the focusing context Ω to the non-focusing context Γ, the synthesis rule has the form:

`<<no parses (char 27): G, x : A ; O async |- C =>+*** t ; D >>`

A not left async

--- $\Uparrow_L^+$

`<<no parses (char 31): G ; {O, x : A} async |- C =>+*** t ; D >>`

By induction on the first premise, we have that:

$$\Gamma, x : A, \Omega \vdash C \Rightarrow^+ t;\ \Delta \tag{ih}$$

from case 2 of the lemma.

3. Case 3. Right Sync:

(a) Case $\otimes^+_R$

In the case of the right synchronous rule for pair introduction, the synthesis rule has the form:

$$\dfrac{\begin{array}{c}\texttt{<<no parses (char 17): G ; . |- A sync =***>+ t1 ; D1 >>}\\ \texttt{<<no parses (char 17): G ; . |- B sync =***>+ t2 ; D2 >>}\end{array}}{\texttt{<<no parses (char 31): G ; . |- \{Tup A B\} sync =***>+ pair t1 t2 ; D1}}$$

By induction on the premises, we have that:

$$\Gamma \vdash A \Rightarrow^+ t_1; \Delta_1 \tag{ih1}$$

$$\Gamma \vdash B \Rightarrow^+ t_2; \Delta_2 \tag{ih2}$$

from case 3 of the lemma. From which, we can construct the following instantiation of the $\otimes^+_R$ synthesis rule in the non-focusing calculus:

$$\dfrac{\Gamma \vdash A \Rightarrow^+ t_1; \Delta_1 \qquad \Gamma \vdash B \Rightarrow^+ t_2; \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^+ (t_1, t_2); \Delta_1 + \Delta_2}\ \mathrm{R}\otimes^+$$

(b) Case $\oplus 1^+_R$ and $\oplus 2^+_R$

In the case of the right synchronous rules for sum introduction, the synthesis rules have the form:

$$\dfrac{\texttt{<<no parses (char 17): G ; . |- A sync =***>+ t ; D >>}}{\texttt{<<no parses (char 26): G ; . |- \{Sum A B\} sync =***>+ inl t ; D >>}}\ \oplus 1^+_L$$

$$\dfrac{\texttt{<<no parses (char 17): G ; . |- B sync =***>+ t ; D >>}}{\texttt{<<no parses (char 26): G ; . |- \{Sum A B\} sync =***>+ inr t ; D >>}}\ \oplus 2^+_L$$

By induction on the premises of the rules, we have that:

$$\Gamma \vdash A \Rightarrow^{+} t;\ \Delta \tag{ih1}$$

$$\Gamma \vdash B \Rightarrow^{+} t;\ \Delta \tag{ih2}$$

from case 3 of the lemma. From which, we can construct the following instantiations of the $\oplus 1_R^+$ and $\oplus 2_R^+$ synthesis rules in the non-focusing calculus, respectively:

$$\dfrac{\Gamma \vdash A \Rightarrow^{+} t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{+} \mathbf{inl}\, t;\ \Delta}\ \mathrm{R}\oplus_1^+$$

$$\dfrac{\Gamma \vdash B \Rightarrow^{+} t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{+} \mathbf{inr}\, t;\ \Delta}\ \mathrm{R}\oplus_2^+$$

(c) Case 1_R^+

In the case of the right synchronous rule for unit introduction, the synthesis rule has the form:

$$\dfrac{}{\texttt{<<no parses (char 15): G ; . |- Unit =***>+ () ; . >>}}\ 1_R^+$$

From which, we can construct the following instantiation of the 1_R^+ synthesis rule in the non-focusing calculus:

$$\dfrac{}{\Gamma \vdash \mathbf{1} \Rightarrow^{+} ();\ \emptyset}\ \mathrm{R}1^+$$

(d) Case $\Box_R^+$

In the case of the right synchronous rule for graded modality introduction, the synthesis rule has the form:

$$\frac{\texttt{<<no parses (char 19): G; . |- A async =>+*** t ; D >>}}{\texttt{<<no parses (char 25): G ; . |- \{ [] r A\} sync =***>+ [t] ; r * D >>}}\ \Box$$

By induction on the premise, we have that:

$$\Gamma \vdash A \Rightarrow^{+} t; \Delta \qquad \text{(ih)}$$

from case 1 of the lemma. From which, we can construct the following instantiation of the $\Box_R^+$ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma \vdash A \Rightarrow^{+} t; \Delta}{\Gamma \vdash \Box_r A \Rightarrow^{+} [t]; r \cdot \Delta}\ \mathrm{R}\Box^{+}$$

(e) Case $\Downarrow_R^+$

In the case of the right synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

$$\frac{\texttt{<<no parses (char 22): G ; . |- A async =>+*** t ; D >>}}{\texttt{<<no parses (char 18): G ; . |- A sync =***>+ t ; D >>}}\ \Downarrow_R^+$$

By induction on the premise, we have that:

$$\Gamma \vdash A \Rightarrow^{+} t; \Delta \qquad \text{(ih)}$$

from case 1 of the lemma.

4. Case 4. Left Sync

(a) Case $\multimap_L^+$

In the case of the left synchronous rule for application, the synthesis rule has the form:

```
                    <<no parses (char 27):  G ; {x2 :  B} sync |- C =>+*** 
                        <<no parses (char 17):  G ; .  |- A sync =***>+
<<no parses (char 35):  G ; { x1 :  A -o B } sync |- C =>+*** [(x1 t2) /
```

By induction on the first premise, we have that:

$$\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1;\, \Delta_1, x_2 : B \tag{ih1}$$

from case 4 of the lemma. By induction on the second premise, we have that:

$$\Gamma \vdash A \Rightarrow^+ t_2;\, \Delta_2 \tag{ih2}$$

from case 3 of the lemma. From which, we can construct the following instantiation of the $\multimap_L^+$synthesis rule in the non-focusing calculus:

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1;\, \Delta_1, x_2 : B \qquad \Gamma \vdash A \Rightarrow^+ t_2;\, \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^+ [(x_1\ t_2)/x_2]t_1;\, (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \text{L}\multimap^+$$

(b) Case LinVar$^+$

In the case of the left synchronous rule for linear variable synthesis, the synthesis rule has the form:

```
                                                                  LinVar+
<<no parses (char 16):  G ; x :  A |- A =***>+ x ; x :  A >>
```

From which, we can construct the following instantiation of the LinVar$^+$

in the non-focusing calculus:

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^{+} x;\ x : A}\ \textsc{LinVar}^{+}$$

(c) Case $\textsc{GrVar}^{+}$

In the case of the left synchronous rule for graded variable synthesis, the synthesis rule has the form:

$$\frac{}{\texttt{<<no parses (char 20): G ; x : [A] r |- A =***>+ x ; x : [A] 1 >>}}\ \textsc{Gr}$$

From which, we can construct the following instantiation of the $\textsc{GrVar}^{+}$ synthesis rule in the non-focusing calculus:

$$\frac{}{\Gamma, x :_{r} A \vdash A \Rightarrow^{+} x;\ x :_{1} A}\ \textsc{GrVar}^{+}$$

(d) Case $\Downarrow_{L}^{+}$

In the case of the left synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

$$\frac{\texttt{<<no parses (char 26): G ; \{x : A\} async |- C =>+*** t ; D >>} \qquad \text{A not atomic and not left sync}}{\texttt{<<no parses (char 25): G ; \{x : A\} sync |- C =>+*** t ; D >>}}\ \Downarrow_{L}^{+}$$

By induction on the premise, we have that:

$$\Gamma, x : A \vdash C \Rightarrow^{+} t;\ \Delta \qquad \text{(ih)}$$

from case 2 of the lemma.

5. Case 5. Focus Right: focus^+_R

   In the case of the focusing rule for transitioning from a left asynchronous judgement to a right synchronous judgement, the synthesis rule has the form:

$$\dfrac{\texttt{<<no parses (char 17): G ; . |- C sync =***>+ t ; D >>} \quad \text{C not atomic}}{\texttt{<<no parses (char 21): G ; . async |- C =>+*** t ; D >>}}\ \textsc{focus}^+_R$$

   By induction on the first premise, we have that:

$$\Gamma \vdash C \Rightarrow^+ t; \Delta \qquad \text{(ih)}$$

   from case 2 of the lemma.

6. Case 6. Focus Left: focus^+_L

   In the case of the focusing rule for transitioning from a left asynchronous judgement to a left synchronous judgement, the synthesis rule has the form:

$$\dfrac{\texttt{<<no parses (char 25): G ; \{x : A\} sync |- C =>+*** t ; D >>}}{\texttt{<<no parses (char 27): G, x : A; . async |- C =>+*** t ; D >>}}\ \textsc{focus}^+_L$$

   By induction on the first premise, we have that:

$$\Gamma, x : A \vdash C \Rightarrow^+ t; \Delta \qquad \text{(ih)}$$

   from case 2 of the lemma.

□

Lemma B.2.7 (Soundness of focusing for additive pruning synthesis)**.** *For all contexts* Γ, Ω *and types* A *then:*

1. *Right Async* : *<<no parses (char 21): G ; O |- A async =>+*** t ; D >>*
2. *Left Async* : *<<no parses (char 21): G ; O async |- C =>+*** t ; D >>*
3. *Right Sync* : *<<no parses (char 18): G ; . |- A sync =***>+ t ; D >>*
4. *Left Sync* : *<<no parses (char 26): G ; {x : A }sync |- C =>+*** t ; D >>*
5. *Focus Right* : *<<no parses (char 21): G ; O async |- C =>+*** t ; D >>*
6. *Focus Left* : *<<no parses (char 28): G, x : A ; O async |- C =>+*** t ; D >*

Proof. 1. Case: 1. Right Async: The proofs for right asynchronous rules are equivalent to those of lemma (B.2.6)

2. Case 2. Left Async: The proofs for left asynchronous rules are equivalent to those of lemma (B.2.6)

3. Case 3. Right Sync: The proofs for right synchronous rules are equivalent to those of lemma (B.2.6), except for the case of the $\otimes'^{+}_{R}$rule:

   (a) Case $\otimes'^{+}_{R}$

   In the case of the right synchronous rule for pair introduction, the synthesis rule has the form:

```
          <<no parses (char 12):  G ; .  |- A =***>+ t1 ; D1 >>
       <<no parses (char 17):  G - D1 ; .  |- B =***>+ t2 ; D2 >>
---------------------------------------------------------------------
<<no parses (char 18):  G ; .  |- Tup A B =***>+ pair t1 t2 ; D1 + D2 >>
```

By induction on the premises, we have that:

$$\Gamma \vdash A \Rightarrow^{+} t_1;\, \Delta_1 \tag{ih1}$$

$$\Gamma - \Delta_1 \vdash B \Rightarrow^{+} t_2;\, \Delta_2 \tag{ih2}$$

from case 3 of the lemma. From which, we can construct the following instantiation of the $\otimes_R'^{+}$ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma \vdash A \Rightarrow^{+} t_1;\, \Delta_1 \qquad \Gamma - \Delta_1 \vdash B \Rightarrow^{+} t_2;\, \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{+} (t_1, t_2);\, \Delta_1 + \Delta_2}\,\mathrm{R}'\otimes^{+}$$

4. Case 4. Left Sync: The proofs for left synchronous rules are equivalent to those of lemma (B.2.6), except for the case of the $\multimap_L'^{+}$ rule:

   (a) Case $\multimap_L'^{+}$

   In the case of the left synchronous rule for application, the synthesis rule has the form:

```
                <<no parses (char 17):  G ; x2 :  B |- C =***>+ t1 ; D1,
                  <<no parses (char 17):  G - D1 ; .  |- A =***>+ t2
<<no parses (char 22):  G ; x1 :  A -o B |- C =***>+ [(x1 t2) / x2] t1 ;
```

   By induction on the first premise, we have that:

$$\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1;\, \Delta_1, x_2 : B \tag{ih1}$$

   from case 4 of the lemma. By induction on the second premise, we have that:

$$\Gamma \vdash A \Rightarrow^{+} t_2;\ \Delta_2 \tag{ih2}$$

from case 3 of the lemma. From which, we can construct the following instantiation of the $\multimap'^{+}_{L}$ synthesis rule in the non-focusing calculus:

$$\dfrac{\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1;\ \Delta_1, x_2 : B \qquad \Gamma - \Delta_1 \vdash A \Rightarrow^{+} t_2;\ \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{+} [(x_1\ t_2)/x_2]t_1;\ (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \mathrm{L}' \multimap^{+}$$

5. Case 5. Right Focus: focus^{+}_{R} - The proof for right focusing rule is equivalent to that of lemma (B.2.6)

6. Case 6. Left Focus: focus^{+}_{L} - The proof for left focusing rule is equivalent to that of lemma (B.2.6)

□