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

June 2023

Abstract

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

Acknowledgements

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Chapter 2

Background

This chapter provides the relevant background information. It largely

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

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

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

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

Table 1: Timeline.

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

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

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

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

2.1 Linear and substructural logics

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

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

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

Linear logic is known as a *substructural* logic because it lacks the weakening and

contraction rules, while permitting exchange.

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

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

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

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

2.1.1 A graded linear typing calculus

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

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

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

The type $\Box_r A$ is an indexed family of type operators where r is a *grade* ranging over the

elements of a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ parameterising the calculus (where $*$ and $+$ are monotonic with respect to the pre-order $\sqsubseteq$).

The syntax of terms provides the elimination and introduction forms:

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

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

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

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

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

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

Definition 2.1.1 (Context addition)**.**

$$(\Gamma, x : A) + \Gamma' = (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma'| \qquad \emptyset + \Gamma = \Gamma$$
$$\Gamma + (\Gamma', x : A) = (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma| \qquad \Gamma + \emptyset = \Gamma$$
$$(\Gamma, x :_r A) + (\Gamma', x :_s A) = (\Gamma + \Gamma'), x :_{(r+s)} A$$

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:

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

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

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

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

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

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

of Γ by r where every assumption in Γ must already be graded (written $[\Gamma]$ in the rule), given by definition (2.1.2).

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

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

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

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

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

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

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 Orchard, Liepelt and III (2019), which is leveraged in soundness of the synthesis calculi.

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

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

2.2 Graded types: An alternative history

JOH: talk about lineage of graded type systems

2.2.1 Graded-base

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

The syntax of graded-base types is given by:

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

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

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

The syntax of terms is given as:

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

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

Definition 2.2.1 (Context addition)**.**

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

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

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

Figure 2: Typing rules for graded-base

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

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

2.3 Two typed calculi

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

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

2.

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

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

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

Chapter 3

A core synthesis calculus

We begin our synthesis journey by designing a synthesis tool for the linear-base target language we introduced in chapter 2. Recall from 1 that we can derive a synthesis judgement as an inversion of our typing judgement. JOH: blah blah blah

To do this we must overcome the problem of *resource management*. This issue was touched on in 1. In section 3.2, we expand on the problem and provide an overview of its history. We identify two feasible approaches to resource management named *additive* and *subtractive*, and implement a synthesis calculus for both in section 3.3. Both approaches follow a similar structure; inverting the typing rules to derive a a set of synthesis rules, yet they differ significantly in how resources are managed.

These differences carry implications with regard to performance and implementation. Both calculi are implemented as part of a synthesis tool for Granule . Having outlined these two solutions to the resource-management problem, we then evaluate the performance of our implementations against a set of benchmarks.

3.1 A practical target language

Our linear base calculus presented in chapter 2 contains only the absolute esssential 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:

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

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.

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

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

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

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

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

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 Orchard, Liepelt and III (2019), 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:

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

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

Example 3.1.2. Graded modalities can capture a form of information-flow security, tracking the flow of labelled data through a program Orchard, Liepelt and III (2019), 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:

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

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* Harland and Pym (2000); Cervesato, Hodas and Pfenning (2000): 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 Hodas and Miller (1994), refined by Cervasto et al. Cervesato, Hodas and Pfenning (2000), 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:

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

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:

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

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 Orchard, Liepelt and III (2019). Hughes and Orchard termed the above approach *subtractive* resource management (in a style similar to *left-over* type-checking for linear type systems Allais (2018); Zalakain and Dardha (2020)). 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 Orchard, Liepelt and III (2019); Gaboardi et al. (2016); **?**, intervals of usage Orchard, Liepelt and III (2019), or hardware schedules Ghica and Smith (2014). 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:

$$\frac{\exists r'.r' + 1 = r \quad \exists s'.s' + 1 = s \qquad x :_r A, y :_s B \vdash A \Rightarrow^- x \mid x :_{r'} A, y :_s B \qquad x :_{r'} A, y :_s B \vdash B \Rightarrow^- y \mid x :_{r'} A, y :_{s'} B}{x :_r A, y :_s B \vdash A \otimes B \Rightarrow^- (x, y) \mid x :_{r'} A, y :_{s'} B} \otimes^-_{\text{INTRO}}$$

(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 Hodas and Miller (1994); Cervesato, Hodas and Pfenning (2000), 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. Appendix A.1.1 collects the rules for reference.

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:

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

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 Hodas and Miller (1994)):

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

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:

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

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:

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

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.

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

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

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 $\sqsubseteq$.

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

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

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:

$$\Gamma, y :_{[0\ldots5]} A', x_2 : A \vdash C \Rightarrow^- t_1 \mid y :_{[2\ldots5]} A'$$
$$\Gamma, y :_{[0\ldots5]} A', x_3 : B \vdash C \Rightarrow^- t_2 \mid y :_{[3\ldots4]} A'$$

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 $\mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2$ the output context must be pessimistic about what resources are left, thus we take the greatest-lower bound yielding the interval $[2 \ldots 4]$ 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:

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

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:*

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

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 **??** *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. 2.2.1) 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:

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

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.

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

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

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

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

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*:*

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

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

JOH: expand here with diagram and full focusing calculi - maybe talk through the subtractive rules JOH: possibly make this a section about implementaiton with a focusing subsection

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 de Moura and Bjørner (2008). We use the LogicT monad for backtracking search Kiselyov et al. (2005) and the Scrap Your Reprinter library for splicing synthesised code into syntactic "holes", preserving the rest of the program text Clarke, Liepelt and Orchard (2017).

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 ! Girard (1987), I/O using ! Liang and Miller (2009), and coKleisli composition over $\mathbb{N}$ and arbitrary semirings: e.g. $\forall r, s \in \mathcal{R}$:

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

- **Dist**: distributive laws of various graded modalities over functions, sums, and products Hughes and Orchard (2020), e.g., $\forall r \in \mathbb{N}$, or $\forall r \in \mathcal{R}$ in any semiring, or $r = [0...\infty]$:

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

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

$$vmap_5 : \Box_5(A \to B) \to ((((A \otimes A) \otimes A) \otimes A) \otimes A) \to ((((B \otimes B) \otimes B) \otimes B) \otimes B)$$

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

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

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

		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

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 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 ($\otimes$-$\mathcal{R}$).

3.6 Conclusion

JOH: set up deriving work: problem with synthesising push + examples

Chapter 4

Deriving graded combinators

Chapter 5

An extended synthesis calculus

Chapter 6

Conclusion

Bibliography

Allais, G. (2018). Typing with Leftovers-A mechanization of Intuitionistic Multiplicative-Additive Linear Logic. In *23rd International Conference on Types for Proofs and Programs (TYPES 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Cervesato, I., Hodas, J. S. and Pfenning, F. (2000). Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1), pp. 133 – 163.

Clarke, H., Liepelt, V.-B. and Orchard, D. (2017). Scrap your reprinter.

de Moura, L. and Bjørner, N. (2008). Z3: an efficient smt solver. pp. 337–340.

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

Ghica, D. R. and Smith, A. I. (2014). Bounded linear types in a resource semiring. In Z. Shao, ed., *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014*, *Lecture Notes in Computer Science*, vol. 8410, Springer, pp. 331–350.

Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, 50(1), pp. 1 – 101.

Harland, J. and Pym, D. J. (2000). Resource-distribution via boolean constraints. *CoRR*, cs.LO/0012018.

Hodas, J. and Miller, D. (1994). Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2), pp. 327 – 365.

Hughes, J. and Orchard, D. (2020). Deriving distributive laws for graded linear types (extended abstract).

Kiselyov, O., Shan, C.-c., Friedman, D. P. and Sabry, A. (2005). Backtracking, interleaving, and terminating monad transformers: (functional pearl). *SIGPLAN Not*, 40(9), p. 192–203.

Liang, C. and Miller, D. (2009). Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46), pp. 4747–4768.

Orchard, D., Liepelt, V. and III, H. E. (2019). Quantitative program reasoning with graded modal types. *PACMPL*, 3(ICFP), pp. 110:1–110:30.

Zalakain, U. and Dardha, O. (2020). Pi with leftovers: a mechanisation in Agda. *arXiv preprint arXiv:200505902*.

Appendix A

Collected synthesis rules

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

A.1.1 Collected rules of the subtractive calculus

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

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

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

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

$$\frac{\Gamma \vdash A \Rightarrow^- t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^- [t] \mid \Gamma - r \cdot (\Gamma - \Delta)}\ \Box^-_R \qquad \frac{\Gamma, x_2 :_r A \vdash B \Rightarrow^- t \mid \Delta, x_2 :_s A \quad 0 \leq s}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^- \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid \Delta}\ \Box^-_L$$

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

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

$$\frac{\Gamma \vdash A \Rightarrow^- t \mid \Delta \qquad\qquad \Gamma \vdash B \Rightarrow^- t \mid \Delta}{\Gamma \vdash A \mathbin{\&} B \Rightarrow^- t \mid \Delta}\ \mathbin{\&}^-_R$$

A.1.2 Collected rules of the additive calculus

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma; \Omega, x : A \vdash B \Uparrow \Rightarrow^- t; \Delta \quad x \notin |\Delta|}{\Gamma; \Omega \vdash A \multimap B \Uparrow \Rightarrow^- \lambda x.t; \Delta} \multimap^-_R \qquad \frac{\Gamma; \Omega \Uparrow \vdash C \Rightarrow^- t; \Delta \quad C \text{ not right async}}{\Gamma; \Omega \vdash C \Uparrow \Rightarrow^- t; \Delta} \Uparrow^-_R$$

LeftAsync

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

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

$$\frac{\Gamma; \Omega, x_2 :_r A \Uparrow \vdash B \Rightarrow^- t; \Delta, x_2 :_s A \quad 0 \leq s}{\Gamma; \Omega, x_1 : \Box_r A \Uparrow \vdash B \Rightarrow^- \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t; \Delta} \Box^-_L$$

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

$$\frac{\Gamma; x :_s A, y : A \Uparrow \vdash B \Rightarrow^- t; \Delta, x :_{s'} A \quad y \notin |\Delta| \quad \exists s.\, r \sqsupseteq s + 1}{\Gamma; x :_r A \Uparrow \vdash B \Rightarrow^- [x/y]t; \Delta, x :_{s'} A} \text{DER}^-$$

$$\frac{\Gamma, x : A; \Omega \Uparrow \vdash C \Rightarrow^- t; \Delta \quad \text{A not left async}}{\Gamma; \Omega, x : A \Uparrow \vdash C \Rightarrow^- t; \Delta} \Uparrow^-_L$$

Focus

$$\frac{\Gamma; \emptyset \vdash C \Downarrow \Rightarrow^- t; \Delta \quad \text{C not atomic}}{\Gamma; \emptyset \Uparrow \vdash C \Rightarrow^- t; \Delta} \text{FOCUS}^-_R \qquad \frac{\Gamma; x : A \Downarrow \vdash C \Rightarrow^- t; \Delta}{\Gamma, x : A; \emptyset \Uparrow \vdash C \Rightarrow^- t; \Delta} \text{FOCUS}^-_L$$

RightSync

$$\frac{\Gamma; \emptyset \vdash A \Downarrow \Rightarrow^- t_1; \Delta_1 \quad \Delta_1; \emptyset \vdash B \Downarrow \Rightarrow^- t_2; \Delta_2}{\Gamma; \emptyset \vdash A \otimes B \Downarrow \Rightarrow^- (t_1, t_2); \Delta_2} \otimes^-_R$$

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

$$\frac{\Gamma; \emptyset \vdash A \Uparrow \Rightarrow^- t; \Delta}{\Gamma; \emptyset \vdash \Box_r A \Downarrow \Rightarrow^- t; \Gamma - r \cdot (\Gamma - \Delta)} \Box^-_R$$

$$\frac{}{\Gamma \vdash 1 \Rightarrow^- () \mid \Gamma} 1^-_R \qquad \frac{\Gamma; \emptyset \vdash A \Uparrow \Rightarrow^- t; \Delta}{\Gamma; \emptyset \vdash A \Downarrow \Rightarrow^- t; \Delta} \Downarrow^-_R$$

A.2.2 Collected rules of the additive focusing calculus

RightAsync

$$\frac{\Gamma; \Omega, x : A \vdash B \Uparrow \Rightarrow^+ t; \Delta, x : A}{\Gamma; \Omega \vdash A \multimap B \Uparrow \Rightarrow^+ \lambda x.t; \Delta} \multimap_R^+ \qquad \frac{\Gamma; \Omega \Uparrow \vdash C \Rightarrow^+ t; \Delta \qquad C \text{ not right async}}{\Gamma; \Omega \vdash C \Uparrow \Rightarrow^+ t; \Delta} \Uparrow_R^+$$

LeftAsync

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

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

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

$$\frac{\Gamma; x :_s A, y : A \Uparrow \vdash B \Rightarrow^+ t; \Delta, y : A}{\Gamma; x :_s A \Uparrow \vdash B \Rightarrow^+ [x/y]t; \Delta + x :_1 A}\ \text{DER}^+ \qquad \frac{\Gamma; \emptyset \vdash C \Rightarrow^+ t; \Delta}{\Gamma; x : 1 \vdash C \Rightarrow^+ \mathbf{let}\,() = x\,\mathbf{in}\,t; \Delta, x : 1} 1_L^+$$

$$\frac{\Gamma, x : A; \Omega \Uparrow \vdash C \Rightarrow^+ t; \Delta \qquad \text{A not left async}}{\Gamma; \Omega, x : A \Uparrow \vdash C \Rightarrow^+ t; \Delta} \Uparrow_L^+$$

Focus

$$\frac{\Gamma; \emptyset \vdash C \Downarrow \Rightarrow^+ t; \Delta \qquad \text{C not atomic}}{\Gamma; \emptyset \Uparrow \vdash C \Rightarrow^+ t; \Delta}\ \text{FOCUS}_R^+ \qquad \frac{\Gamma; x : A \Downarrow \vdash C \Rightarrow^+ t; \Delta}{\Gamma, x : A; \emptyset \Uparrow \vdash C \Rightarrow^+ t; \Delta}\ \text{FOCUS}_L^+$$

RightSync

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

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

$$\frac{\Gamma; \emptyset \vdash A \Uparrow \Rightarrow^+ t; \Delta}{\Gamma; \emptyset \vdash \Box_r A \Downarrow \Rightarrow^+ [t]; r \cdot \Delta} \Box_R^+ \qquad \frac{}{\Gamma; \emptyset \vdash 1 \Rightarrow^+ (); \emptyset} 1_R^+$$

$$\frac{\Gamma; \emptyset \vdash A \Uparrow \Rightarrow^+ t; \Delta}{\Gamma; \emptyset \vdash A \Downarrow \Rightarrow^+ t; \Delta} \Downarrow_R^+$$

LeftSync

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

Alternative additive focusing pruning rules for pair introduction and application

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

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

Appendix B

Benchmark problems

B.1 List of linear-base synthesis benchmark problems

Appendix C

Proofs