JACK OLIVER HUGHES

PROGRAM SYNTHESIS FROM LINEAR AND GRADED TYPES

PROGRAM SYNTHESIS FROM LINEAR AND GRADED TYPES

JACK OLIVER HUGHES

University of Kent

A Dissertation in Computer Science

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.

PUBLICATIONS

In this thesis, the content of some chapters is formed from previously published papers:

ACKNOWLEDGMENTS

Put your acknowledgments here.

CONTENTS

LIST OF FIGURES

Part I

PROGRAM SYNTHESIS FROM LINEAR AND GRADED TYPES

1

INTRODUCTION

One of the most useful and well-studied tools available to modern programmers is the type system. Not only do type systems allow many kinds of errors to be caught statically, they also help inform the design of a program. Many programmers will often tend to begin writing their programs by first defining the types, from which the program code follows naturally. This phenomenon will be familiar to any who have written programs in typed functional programming languages, and results from the fact that types form a high-level abstract specification of program behaviour.

Type-directed program synthesis is a well-studied technique for automatically generating program code from a given type specification - the *goal* type. This approach has a long history, which is deeply intertwined with automated theorem proving, thanks to the Curry-Howard correspondence [26, 38].

One lens through which we can view this task is as an inversion of type checking: we start with a goal type and inductively synthesise well-typd sub-terms by breaking the goal into sub-goals, pruning the search space of programs via typing as we go. This approach follows the treatment of program synthesis as a form of proof search in logic: given a type A we want to find a program term t which inhabits A. We can express this in terms of a synthesis *judgement* which acts as a kind of inversion of typing or proof rules:

$$\Gamma \vdash A \Rightarrow t$$

meaning that the term t can be synthesised for the goal type A under a context of assumptions Γ. We may construct a calculus of synthesis *rules* for a programming language, inductively defining the above synthesis judgement for each type former.

For example, we may define a rule for standard product types in the following way:

$$\frac{\Gamma \vdash A \Rightarrow t_1 \qquad \Gamma \vdash B \Rightarrow t_2}{\Gamma \vdash A \times B \Rightarrow (t_1, t_2)}\ \times_{\textsc{Intro}}$$

Reading 'clockwise' from the bottom-left: to synthesise a value of type $A \times B$, we synthesise a value of type A and then a value of type B and combine them into a pair in the conclusion. The 'ingredients' for synthesising the subterms t_1 and t_2 come from the free-variable assumptions Γ and any constructors of A and B, assuming these types are monomorphic.

Depending on the context, there could be many possible combinations of assumption choices to synthesise such a pair. Consider the following partial program containing a program *hole*, marked with ?, specifying a position at which we wish to perform synthesis:

$$\begin{aligned}&f : A \to A \to A \to A \times A\\ &f\ x\ y\ z = ?\end{aligned}$$

The function has three parameters all of type A which can be used to synthesise an expression of the goal type $A \times A$. Expressing this synthesis problem as an instantiation of the above $\times_{\textsc{Intro}}$ rule:

$$\frac{x : A, y : A, z : A \vdash A \Rightarrow t_1 \qquad x : A, y : A, z : A \vdash A \Rightarrow t_2}{x : A, y : A, z : A \vdash A \times A \Rightarrow (t_1, t_2)}\ \times_{\textsc{Intro}}$$

Even in this simple setting, the number of possibilities starts to become unwieldy: there are nine (3^2) possible candidate programs based on combinations of x, y and z. Ideally, we would like some way of constraining the number of choices that are required by the synthesis algorithm. Many systems achieve this by allowing the user to specify additional information about their desired program behaviour. For example, recent work has extended type-directed synthesis to refinement types [46], cost specifications [34], differential privacy [48], example-guided synthesis [2, 18] or examples integrated with types Frankle et al. [20] and Osera and Zdancewic [44], and ownership information [19]. The general idea is that, with more information, whether that be richer types, additional examples, or behavioural specifications, the proof search / program synthesis process can be pruned and refined.

This work presents a program synthesis approach that leverages the information contained in *linear* and *graded type systems* that track and enforce program properties related to data flow, statically. We refer to these systems as *resourceful* type systems, since they treat data as though it is a physical resource, constraining how data can be used by a program and thus reducing the number of possible synthesis choices that need to be made. Our hypothesis is that resource-and-type-directed synthesis speeds up type-directed synthesis, reducing the number of paths that need to be explored and the amount of additional specification (e.g. input-output examples) required.

Graded type systems trace their roots to linear logic. In linear logic, data is treated as though it were a finite resource which must be consumed exactly once with arbitrary copying and discarding disallowed [24]. Non-linear use of data is expressed through the ! modal operator (the *exponential modality*). This gives a binary view—a value may either be used exactly once or in a completely unconstrained way. Bounded Linear Logic (BLL) refines this view, replacing ! with a family of indexed modal operators where the index provides an upper bound on usage [25], e.g., $!_{\leq 4}A$ represents a value A which may be used up to 4 times. In recent years, various works have generalised BLL, resulting in *graded* type systems in which these indices are drawn from an arbitrary pre-ordered semiring [1, 5, 10, 14, 23, 39, 45]. This allows numerous program properties to be tracked and enforced statically, including various kinds of reuse, privacy and confidentiality, and capabilities. Such systems are increasingly popular and form the basis of Linear Haskell [7], Idris 2 [8], as well as the experimental programming language Granule [43].

Returning to our example in a graded setting, the arguments of the function now have *grades* (annotations) that, in this context, are natural numbers describing the exact number of times the parameters must be used (the choice here was ours):

$$\begin{aligned} & f : A^2 \rightarrow A^0 \rightarrow A^0 \rightarrow A \times A \\ & f\ x\ y\ z = ? \end{aligned}$$

The first A is annotated with a grade 2, which in this context indicates that it *must* be used twice. Likewise, the types of y and z are graded with 0, enforcing zero usage, i.e., we are not allowed to use them in the body of f and must discard them.

The result is that there is only one (normal form) inhabitant for this type: (x, x); the other assumptions will not even be considered in synthesis, allowing us to effectively prune out branches which use resources in a way which violates the grades. In this example, these annotations take the form of natural numbers explaining how many times a value can be used, but we may instead wish to represent different kinds of program properties, such as sensitivity, strictness, or security levels for tracking non-interference, all of which are well-known instances of graded type systems [1, 21, 43]. Note that all of these examples are technically graded presentations of *coeffects*, tracking how a programs uses its context, in contrast with graded types for side *effects* [32, 42], which we do not consider here.

1.1 CONTRIBUTIONS

The primary aim of this work is to demonstrate the feasibility and power of using resourceful types as the basis of a type-directed program synthesis tool, culminating in the development and implementation of an expressive, efficient, and feature-rich program synthesis tool for the program language Granule [43]. Granule is a functional programming language which combines linear, graded, and indexed data types; although we concern ourselves only with the former two in this work.

Specifically, this work makes the following contributions:

- We identify two approaches which make type-directed program synthesis in a resourceful setting feasible. Drawing inspiration from the work of Hodas and Miller on theorem proving [29], we adapt their method to graded types, and propose a dual approach, yielding two strategies for managing the usage of values as resources in the synthesis of a program term.

- We use these approaches to construct two simple synthesis calculi for a simplified core of Granule, which demonstrate their effectiveness as tools for resourceful program synthesis. We implement both approaches as part of the Granule toolchain.

- We showcase an alternative and complementary approach to generating a subset of Granule programs, making use of a system inspired by Haskell's deriving mechanism [37] adapted to graded types.

- We then define a synthesis calculus for a fully graded type system. This type system is a feature-rich language based on Granule's *graded base* language extension, which includes recursion, recursive types, and user-defined ADTs. Furthermore, we again implement this calculus as part of the Granule toolchain.

- We evaluate our tool on a benchmark suite of recursive functional programs leveraging standard data types like lists, streams, and trees. We compare against non-graded synthesis provided by Myth [44].

- We prove that each of these systems is sound, i.e., synthesised programs are typed by the goal type. A ket property here is that synthesised programs are not well-typed, but also *well-resourced*, meaning that all values inside the program are used according to their resource constraints. We show that this property holds for each of our synthesis calculi as part of their soundness proofs.

- We demonstrate how our approach to resourceful program synthesis can be readily applied to other graded systems. Leveraging our calculus and implementation, we provide a prototype tool for synthesising Haskell programs written using GHC 9's linear types extension.

1.2 STRUCTURE

This dissertation is structured into six chapters. In the next chapter, Chapter 2, the theoretical background of linear and graded types is laid out. In doing so, we introduce two core calculi with simple types and grades, which demonstrate the two major lineages of resourceful type systems. The first is a language based on an underlying non-graded type system (in this case a linear type system), with graded modal types introduced and eliminated explicitly. This system is the default basis of Granule [43], the target language of our implementation. The second caculus does away with this linear basis, embedding graded modalities into the function types a la Idris 2 and Linear Haskell. McBride's QTT [5, 39], the core of Linear Haskell [7], and the unified graded modal calculus of [1]. This system is also present in Granule in the form of an optional language extension.

The rest of the dissertation is structured such that synthesis calculi for both of these systems are defined and presented, minimising any redundancy in the presentation. Despite the variances between the core calculi of 2, there is a substantial degree of overlap between the two. Thus, we adopt the following structure:

1. Chapter 3 introduces the core concepts of type-directed program synthesis from resourceful type systems using an extenion of the typing calculus of section 2.3. Specifically this chapter introduces the *resource management problem* as it relates to program synthesis: how do we ensure that a synthesised program is not only well-typed but also well-resourced? To address this question, we define two calculi of synthesis rules based on the graded linear λ-calculus which tackle the problem in different ways. To better illustrate and test the practicality of the synthesis calculi, we extend the language with multiplicative conjunction (product types $\otimes$ and unit 1) and additive disjunction (sum types $\otimes$). These calculi are then implemented targeting deafult Granule.

   We produce a comparative evaluation of the implemented synthesis tool, contrasting the resource management approaches against each other, before selecting the most performant to use going forward.

2. We then make a brief diversion from type-directed synthesis, exploring a mechanism for automatically deriving programs from their type à la Haskell's generic deriving mechanism [37]. Again for this we base the approach on the graded linear λ-calculus, extending it further with data constructors, pattern matching, and recursive data types.

3. Finally, in chapter 5, we present a synthesis calculus for a target language based instead on the core graded λ-calculus of 2.4. This calculus incorporates all of the language features that have been introduced in the previous chapters, for a rich synthesis tool implementation targeting Granule's *graded base* language extension. Furthermore, we outline several other useful extensions to the synthesis tool, such as the inclusion of example-based synthesis, and a post-synthesis refactoring process which re-writes synthesised programs in a more idiomatic style.

> We then evaluate the implementation on a set of 46 benchmarks, including several non-trivial programs which make use of these new features. In this evaluation, we compare to From our evaluation we find that using grades in synthesis outperforms purely type-driven program synthesis in terms of both speed, number of input-output examples required or number of retries to get the desired program.
>
> Finally, to demonstrate the practicality and versatility of our approach, we apply our synthesis algorithm of Chapter 5 to synthesising programs in Haskell from type signatures using GHC's *linear types* extension (which is implemented underneath by a graded type system).

This approach strikes a balance between maximising coverage of different approaches to resourceful type systems, and avoiding uneccessary repetition, whilst gradually increasing the complexity of the target language. By the end, we will then have two synthesis tool implementations for Granule, targeting both styles of graded type systems.

2

BACKGROUND

Since Girard's original work on Linear Logic in 1987 [24], the development of type systems which convey additional information about the program's structure has evolved into a distinct paradigm, culminating in recent years with the notion of *graded types*. Approaches to graded type systems run the gamut, incorporating a wide range of effect and coeffect systems, however, they can typically be distilled into two categories, with distinct lineages:

- Systems where a graded modal type operator introduces and eliminates graded modalities above some existing type system. This is the default approach of Granule, where the underlying type system is linear, and grade modalities are introduced and eliminated via the $\Box$ modal type operator.

- Systems where grades permeate the program, and are introduced via annotations on function arrows. This is the approach taken by Linear Haskell [7], where grades (or "multiplicities") are specified using the `%` operator.

These two different styles to graded types mirror the dual development of effect systems and graded monadic systems in the literature. In the latter case, the two were eventually found to be equivalent.

2.1 TERMINOLOGY

Before delving into linear and graded systems, we briefly frame the approach we will take to discussing the relevant background material. Throughout 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. 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*.

2.2 LINEAR AND SUBSTRUCTURAL LOGICS

Linear logic [**<empty citation>**] was introduced 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.

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

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

Figure 2.1: Substructural rules for weakening, contraction, and 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 [**<empty citation>**]. 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.

2.3 THE GRADED LINEAR λ-CALCULUS

We now define a core type system, based on the linear λ-calculus extended with a graded modal type. This calculus is equivalent to the core calculus of Granule, GrMini[**<empty citation>**]. Granule's full type system extends this graded linear core with polymorphism, algebraic data types (ADTs), indexed types, pattern matching. We refer to this system as the *graded linear λ-calculus*, reflecting the underlying linear structure of the system.

This system forms the basis of the target language for our synthesis tool in chapter 3.

The types of the graded linear λ-calculus are defined as:

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

where 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 $\sqsubseteq$). Linear functions are specified by the $\multimap$ arrow[1].

[1] Although Granule uses $\rightarrow$ syntax rather than $\multimap$ for linear functions for the sake of familiarity with standard functional languages

The syntax of terms provides the elimination and introduction forms:

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

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

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

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

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

Various operations on contexts are used to capture non-linear data flow via grading. Firstly, *context addition* (2.3.1) 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 2.3.1 (Context addition)**.**

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

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:

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

$$\frac{}{x : A \vdash x : A}\,\textsc{Var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\,\textsc{Abs}$$

$$\frac{\Gamma_1 \vdash t_1 : A \to B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\,\textsc{App}$$

$$\frac{\Gamma \vdash t : A}{\Gamma, [\Delta]_0 \vdash t : A}\,\textsc{Weak} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x :_1 A \vdash t : B}\,\textsc{Der}$$

$$\frac{\Gamma, x :_r A, \Gamma' \vdash t : A \quad r \sqsubseteq s}{\Gamma, x :_s A, \Gamma' \vdash t : A}\,\textsc{Approx}$$

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

Figure 2.2: Typing rules of the graded linear λ-calculus

Figure 5.1 defines the typing rules. Linear variables are typed in a singleton context (Var). Abstraction (Abs) and application (App) follow the rules of the linear λ-calculus. The Weak rule captures weakening of assumptions graded by 0 (where $[\Delta]_0$ denotes a context containing only graded assumptions graded by 0). 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 (2.3.2).

Definition 2.3.2 (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 2.3.1. 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 [43], which is leveraged in soundness of the synthesis calculi.

Lemma 2.3.1 (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 $\Gamma + (r \cdot \Delta) + \Gamma' \vdash [t'/x]t : B$*

2.4 THE FULLY GRADED λ-CALCULUS

We now define a core calculus for a fully graded type system, in the vein of systems where grades permeate the entire program, drawing from the coeffect calculus of Petricek, Orchard, and Mycroft [45], Quantitative Type Theory (QTT) by McBride [39] and refined further by Atkey [5] (although we omit dependent types from our language), the calculus of Abel and Bernardy [1], and other graded dependent type theories [5, 40]. Similar systems also form the basis of the core of the linear types extension to Haskell [7]. We refer to this system as the *fully graded λ-calculus* to differentiate it from its linearly-based counterpart.

The syntax of graded-base types is given by:

$$A, B ::= A^r \rightarrow B \mid \Box_r A \qquad (types)$$

where the function arrow $A^r \rightarrow B$ annotates the input type with a *grade* r which is again drawn from a pre-ordered semiring

$(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ paramterising the calculus. 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:

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

Terms comprise the λ-calculus, extended with the *promotion* construct [t] as seen in section 2.3. Typing judgements have the same form as section 2.3, however, variable contexts are instead 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 $x :_r A$, which must be used in a way which adheres to the constraints of the grade r. As before, structural exchange is permitted, allowing a context to be arbitrarily reordered.

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

$$\frac{\Gamma_1 \vdash t_1 : A^r \to B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + r \cdot \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{App}$$

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

Figure 2.3: Typing rules for graded-base

Figure 2.3 gives the full typing rules, which explains the meaning of the syntax with reference to their static semantics.

Variables (rule Var) are typed in a context where the variable x has grade 1 denoting its single usage here. All other variable assumptions are given the grade of the 0 semiring element (providing *weakening*), using *scalar multiplication* of contexts by a grade, re-using definition (2.3.2).

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 2.4.1 (Graded context addition)**.**

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

Note that 2.4.1 differs only from 2.3.1, in that the former need not consider linear assumptions.

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, the pre-order relation of the semiring $\sqsubseteq$ provides approximation. 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 $r \sqsubseteq r'$.

METATHEORY Lastly we note that the fully graded λ-calculus also enjoys admissibility of substitution [1] which is critical in type preservation proofs, and is needed in our proof of soundness for synthesis:

Lemma 2.4.1 (Admissibility of substitution)**.** *Let* $\Delta \vdash t' : A$*, then: If* $\Gamma, x :_r A, \Gamma' \vdash t : B$ *then* $\Gamma + (r \cdot \Delta) + \Gamma' \vdash [t'/x]t : B$

2.5 TWO TYPING CALCULI

Having outlined the two lineages of graded type systems, we are left with the question: what approach should we use as the basis of a target language for a program synthesis tool? Both systems embed properties for reasoning about program structure into the language, however, they differ in how this information is expressed, as shown by the variance in typing and syntax between sections 2.3 and 2.4.

Rather than focus entirely on one approach, we opt to instead build synthesis tools which target both languages. As we have seen, systems based on both approaches are used by many languages today, and both pose their unique challenges in designing a synthesis tool, which makes favouring a particular approach difficult to justify. Furthermore, the target programming language Granule of our implementations includes both approaches [2].

[2] As of Granule v0.9.3.0

3

A CORE SYNTHESIS CALCULUS

We begin our first exploration into program synthesis with a system for the graded linear λ-calculus of section 2.3. The primary aim of this chapter is to introduce the core concepts of type-directed program synthesis in a resourceful setting, in particular, the problem of *resource mangement*. We therefore prioritise simplicity over expressivity for our target language, with the core typing calculus of section 2.3 forming an ideal candidate.

As mentioned in chapter 1, type-directed program synthesis can be framed as an inversion of type checking. In type checking, we have a judgement of the form:

$$\Gamma \vdash t : A \qquad \text{(type checking)}$$

whch states that under some context of assumptions Γ we can assign the program term t the type A. Here, Γ and t constitute the "inputs" to the judgement, while the type A forms the "output". Synthesis inverts this judgement form, leaving us with a *synthesis judgement* form:

$$\Gamma \vdash A \Rightarrow t \qquad \text{(synthesis)}$$

which states that we can construct a program term t from the type A, using the assumptions in Γ. As in type checking, Γ forms an input to the judgement. However, A and t exchange roles: the former is now also an input, while the latter is the judgement's output. Program synthesis then becomes a task of inductively enumerating programs in a "bottom-up" starting from the goal type A: A is broken into sub-goals, from which sub-terms are synthesised until the goal can not be broken into further sub-goals. At this point, we either synthesise a usage of a variable from Γ if possible, or synthesis fails. This is the essence of type-directed program synthesis.

Resourceful types introduce another dimension to synthesis: how do we ensure that the assumptions in Γ are used according to their resource constraints in the synthesised term t? I.e. if $x : A$ is a linear assumption in Γ that is used in some way to construct t, then the synthesis algorithm must synthesise a t which uses x exactly once. Likewise, if $x :_r A$ is a graded assumption, then it must be used in t in a way which satisifise its grade r.

This problem has been explored before in the context of automated theorem proving for linear logic, and has been termed the *resource management problem*. We describe this problem in detail in section 3.2 and propose two candidate solutions, basing our approach on the *input-output context management* model described by Hodas and Miller [29], and further developed by Cervesato et al. [11].

The challenges posed by ensuring the well-resourcedness of synthesised programs are most exemplified by the inclusion in our target language of multiplicative conjunction, and additive disjunction. Therefore, prior to fully describing the problem of resource management and our proposed solutions, we first expand our target language with multiplicative product ($\otimes$), and unit types (1), as well as disjunctive sum types ($\oplus$). These extensions are detailed in section 3.1, which will be the target language of the synthesis calculi of this chapter. As well as helping to conceptualise the challenges posed by resourceful program synthesis, these have the added benefit of allowing the synthesis of more expressive programs, without introducing uneccessary complexity at this stage.

Having outlined both a suitable target language and two approaches to dealing with the issue of resource management, we then present two synthesis calculi in section 3.3 as augmented inversions of the typing rules. Each calculus is based on a one of the proposed solutions to the resource management problem, which we then evaluate and contrast against each other in section 3.5.

Both calculi are implemented as part of a synthesis tool for Granule [1]. The implementation is a fairly direct translation of the synthesis calculi into Haskell. We thus elide the details of the implementation, focusing only on an important optimisation technique in section 3.4: focusing. Focusing removes much of the uneccessary non-determinism present in our synthesis rules

[1] The exact implementation of the rules as they stand is deprecated, but may be found in Granule release v0.7.8.0: https://github.com/granule-project/granule/releases/tag/v0.7.8.0

by fixing an ordering on the application of rules. We present the two *focused* forms of our original synthesis calculi which comprise the basis of our Granule implementation.

3.1 A CORE TARGET LANGUAGE

The syntax for the full language is given by the following grammar:

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

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.

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

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

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

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

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

Figure 3.1: Typing rules of for $\otimes$, $\oplus$, and 1

Figure 3.1 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 [43], 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:

$$\begin{aligned}
&\oplus_e : \Box_{0...1}(A \multimap C) \multimap \Box_{0...1}(B \multimap C) \multimap (A \oplus B) \multimap C\\
&\oplus_e = \lambda x'.\lambda y'.\lambda z.\mathbf{let}\ [x] = x'\ \mathbf{in}\\
&\qquad\qquad\qquad\quad \mathbf{let}\ [y] = y'\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad \mathbf{case}\ z\ \mathbf{of}\ \mathbf{inl}\ u \rightarrow x\ u \mid \mathbf{inr}\ v\ \rightarrow y\ v
\end{aligned}$$

Example 3.1.2. Graded modalities can capture a form of information-flow security, tracking the flow of labelled data through a program [43], 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:

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

3.2 THE RESOURCE MANAGEMENT PROBLEM

In chapter 1 we considered a synthesis rule for pairs and highlighted how graded types could be use to control the number of times assumptions are used in the synthesised term.

Chapter 1 considered (Cartesian) product types $\times$, but in our target language we use the multiplicative product of linear types, given in Figure 3.1Each 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 $\otimes$<span>Intro</span> 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 [29], refined by Cervasto et al. [11], 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:

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

(example)

Thus this approach neatly avoids the problem of having to split the input context, and facilitates efficient proof search for linear types. We extend this input-output context management model to graded types to graded types to facilitate the synthesis of programs in Granule. We term the above approach *subtractive* resource management (in a style similar to *left-over* type-checking for linear type systems [3, 51]).

Graded type systems, as we consider them here, have typing contexts in which free-variables are assigned a type, and a grade. 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:

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

(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. These constraints can be discharged via an off-the-shelf SMT solver, such as Z3 [41]. Such calls to an external solver are costly, however, and thus efficiency of resource management is a key concern.

We propose a dual approach to the subtractive: the *additive* resource management scheme. In the additive approach, output contexts describe what was *used* not what was is *left*. In the case of synthesising a term with multiple subterms (like pairs), the output context from each premise is then added together using the semiring addition operation applied pointwise on contexts to produce the final output in the conclusion. For pairs this looks like:

$$\dfrac{\Gamma \vdash A \Rightarrow^{+} t_1 \mid \Delta_1 \qquad \Gamma \vdash B \Rightarrow^{+} t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{+} (t_1, t_2) \mid \Delta_1 + \Delta_2} \otimes^{+}_{\textsc{Intro}}$$

The entirety of Γ is used to synthesise both premises. For example, for a goal of $A \otimes A$:

$$\dfrac{\begin{array}{c} x :_{r} A, y :_{s} B \vdash A \Rightarrow^{+} x \mid x :_{1} A, y :_{0} B \\ x :_{r} A, y :_{s} B \vdash A \Rightarrow^{+} x \mid x :_{1} A, y :_{0} B \end{array}}{x :_{r} A, y :_{s} B \vdash A \otimes A \Rightarrow^{+} (x, x) \mid x :_{1+1} A, y :_{0} B} \otimes^{+}_{\textsc{Intro}}$$

(example)

Later checks in synthesis then determine whether the output context describes usage that is within the grades given by Γ, i.e., that the synthesised terms are *well-resourced*.

Both the subtractive and additive approaches avoid having to split the incoming context Γ into two prior to synthesising subterms.

We adapt the input-output context management model of linear logic synthesis to graded types, pruning the search space via the quantitative constraints of grades. We implement synthesis calculi based on both the additive and subtractive approaches, evaluating their performance on a set of benchmarking synthesis problems.

3.2.1 *Related Work*

Before Hodas and Miller [29], the problem of resource non-determinism was first identified by Harland and Pym [27]. Their solution delays splitting of contexts at a multiplicative connective. They later explored the implementation details of this approach, proposing a solution where proof search is formulated in terms of constraints on propositions. Propositions which occur in the conclusion of a multiplicative connective are assigned a Boolean expression whose solution Constraints generated during the proof search, with a solution to these constituting a valid proof [27]. The logic programming language Lygon [36] implements this approach.

Our approach to synthesis implements a *backward* style of proof search: starting from the goal, recursively search for solutions to subgoals. In contrast to this, *forward* reasoning approaches attempt to reach the goal by building subgoals from previously proved subgoals until the overall goal is proved. Pfenning and Chaudhuri consider forward approaches to proof search in linear logic using the *inverse method* [16] where the issue of resource non-determinism that is typical to backward approaches is absent [12, 13].

3.3 THE SYNTHESIS CALCULI

We now present two synthesis calculi based on the subtractive and additive resource management schemes, respectively. 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.

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{}{\Gamma \vdash 1 \Rightarrow^{-} () \mid \Gamma}\;1_R^{-}$$

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

Figure 3.2: Collected rules of the subtractive synthesis calculus

3.3.1 *Subtractive Resource Management*

Our subtractive approach follows the philosophy of earlier work on linear logic proof search [11, 29], 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 3.2 collects the rules for reference.

3.3.1.1 *Variables*

Variable terms can be synthesised from assumptions in Γ by rules:

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

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

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.

3.3.1.2 *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 [29]):

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

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.

3.3.1.3 *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:

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

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.

3.3.1.4 *Graded modalities*

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

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

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 variable 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:

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

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.

3.3.1.5 *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.

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

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

3.3.1.6 *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:

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

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 $\mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2$ 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.

3.3.1.7 *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:*

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

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* **??** *provides the proof.*

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

$$\frac{}{\Gamma, x :_{r} A \vdash A \Rightarrow^{+} x;\; x :_{1} A}\;\textsc{GrVar}^{+}$$

$$\frac{\Gamma, x :_{s} A, y : A \vdash B \Rightarrow^{+} t;\; \Delta, y : A}{\Gamma, x :_{s} A \vdash B \Rightarrow^{+} [x/y]t;\; \Delta + x :_{1} A}\;\textsc{der}^{+}$$

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

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

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

$$\frac{\begin{array}{c}\Gamma, x_2 :_{r} A \vdash B \Rightarrow^{+} t;\; \Delta \\ \textit{if}\; x_2 :_{s} A \in \Delta \;\textit{then}\; s \sqsubseteq r \;\textit{else}\; 0 \sqsubseteq r\end{array}}{\Gamma, x_1 : \Box_{r} A \vdash B \Rightarrow^{+} \textbf{let}\,[x_2] = x_1\,\textbf{in}\, t;\; (\Delta \backslash x_2), x_1 : \Box_{r} A}\;\Box_{L}^{+}$$

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

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

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

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

$$\frac{\begin{array}{c}\Gamma, x_2 : A \vdash C \Rightarrow^{+} t_1;\; \Delta_1, x_2 : A \\ \Gamma, x_3 : B \vdash C \Rightarrow^{+} t_2;\; \Delta_2, x_3 : B\end{array}}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^{-} \textbf{case}\; x_1 \;\textbf{of inl}\; x_2 \to t_1;\; \textbf{inr}\; x_3 \to t_2 \mid \Delta_1 \sqcup \Delta_2, x_1 : A \oplus B}\;\oplus_{L}^{+}$$

$$\frac{}{\Gamma \vdash 1 \Rightarrow^{+} ();\; \varnothing}\;1_{R}^{+}$$

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

Figure 3.3: Collected rules of the additive synthesis calculus

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.

3.3.2.1 *Variables*

We unpack the rules, starting with variables:

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

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

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.

3.3.2.2 *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:

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

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 $[t]$:

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

3.3.2.3 *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. 2.3.1) 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.

3.3.2.4 *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).

3.3.2.5 *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.

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

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

3.3.2.6 *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.

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

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

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

3.3.2.7 *Unit*

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

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

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

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.

3.3.2.8 *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:

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

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:

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

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

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

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

3.4 FOCUSING

The additive and subtractive calculi presented in sections 3.3.1 and 3.3.2 provide the foundations for the implementations of a synthesis tool for Granule programs. Implementing the rules as they currently stand, however, would yield a highly inefficient tool. In their current form, the rules of both calculi exhibit a high degree of non-determinism with regard to order in which rules can be applied.

This leads to us exploring a large number of redundant search branches: something which can be avoided through the application of a technique from linear logic proof theory called *focusing* [4]. Focusing is based on the observation that some of the synthesis rules are invertible, i.e. whenever the conclusion of the rule is derivable, then so are its premises. In other words, the order in which we apply invertible rules doesn't matter. By fixing a particular ordering on the application of these 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 take our both of our calculi and apply this focusing technique to them, yielding two *focusing* calculi. To do so, we augment our previous synthesis judgement with an additional input context Ω:

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

Unlike Γ and Δ, Ω is an *ordered* context, which behaves like a stack.

Using the terminology of Andreoli, we refer to rules that are invertible as *asynchronous* and rules that are not as *synchronous*. The intuition is that of asynchronous communication: 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, which refines the focusing judgement above with an arrow indicating which part of the judgement is currently in focus. An $\Uparrow$ indicates an asynchronous phase, while a $\Downarrow$ indicates a synchronous (focused) phase. The location of the arrow in the judgement indicates whether we are focusing on the left or right:

1. Right Async: $\multimap_R$ rule with the judgement:

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

2. Left Async: $\otimes_L$, $\oplus_L$, 1_L, Der, and $\square_L$ rules with the judgement:

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

3. Right Sync: $\otimes_R$, $\oplus 1_R$, $\oplus 2_R$, 1_R, and $\square_R$ rules with the judgement:

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

4. Left Sync: $\multimap_L$ rule with the judgement:

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

The complete calculi of focusing synthesis rules are given in Figures 3.4-3.8 for the subtractive calculus, and 3.9-3.13 for the additive, divided into focusing phases. The focusing rules for the additive pruning calculus are identical to the additive calculus, save for the $\otimes_R^+$ and $\multimap_L^+$ rules, which are given in Figure 3.14.

For the most part, the translation from non-focused to focused rules is straightforward. The most notable change occurs in rules in which assumptions are bound. In the cases where a fresh assumption's type falls into the Left Async category (i.e. $\otimes$, $\oplus$, etc.), then it is bound in the ordered context Ω instead of Γ. Left Async rules operate on assumptions in Ω, rather than Γ. This results in invertible elimination rules being applied as fully as possible before *focusing* on non-invertible rules when Ω is empty.

In addition to the focused forms of the original synthesis calculi, each calculus has a set of rules which determine which part of the synthesis judgement will be focused on: the Focus rules. These rules are given by Figures 3.6, and 3.11 for the subtractive and additive calculi, respectively.

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

$$\frac{\Gamma;\Omega \Uparrow \vdash C \Rightarrow^{-} t \mid \Delta \qquad C \text{ not right async}}{\Gamma;\Omega \vdash C \Uparrow \Rightarrow^{-} t \mid \Delta}\ \Uparrow^{-}_{R}$$

Figure 3.4: Right Async rules of the focused subtractive synthesis calculus

$$\frac{\begin{array}{c}\Gamma;\Omega, x_1 : A, x_2 : B \Uparrow \vdash C \Rightarrow^{-} t_2 \mid \Delta \\ x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|\end{array}}{\Gamma;\Omega, x_3 : A \otimes B \Uparrow \vdash C \Rightarrow^{-} \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta}\ \otimes^{-}_{L}$$

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

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

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

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

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

Figure 3.5: Left Async rules of the focused subtractive synthesis calculus

$$\frac{\Gamma;\varnothing \vdash C \Downarrow \Rightarrow^{-} t \mid \Delta \qquad C \text{ not atomic}}{\Gamma;\varnothing \Uparrow \vdash C \Rightarrow^{-} t \mid \Delta}\ \textsc{focus}^{-}_{R}$$

$$\frac{\Gamma; x : A \Downarrow \vdash C \Rightarrow^{-} t \mid \Delta}{\Gamma, x : A;\varnothing \Uparrow \vdash C \Rightarrow^{-} t \mid \Delta}\ \textsc{focus}^{-}_{L}$$

Figure 3.6: Focus rules of the focused subtractive synthesis calculus

$$\dfrac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow^{-} t_1 \mid \Delta_1 \qquad \Delta_1;\varnothing \vdash B \Downarrow \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma;\varnothing \vdash A \otimes B \Downarrow \Rightarrow^{-} (t_1, t_2) \mid \Delta_2}\;\otimes^{-}_{R}$$

$$\dfrac{\Gamma;\varnothing \vdash B \Downarrow \Rightarrow^{-} t \mid \Delta}{\Gamma;\varnothing \vdash A \oplus B \Downarrow \Rightarrow^{-} \mathbf{inr}\, t \mid \Delta}\;\oplus 2^{+}_{L} \quad \dfrac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow^{-} t \mid \Delta}{\Gamma;\varnothing \vdash A \oplus B \Downarrow \Rightarrow^{-} \mathbf{inl}\, t \mid \Delta}\;\oplus 1^{+}_{L}$$

$$\dfrac{\Gamma;\varnothing \vdash A \Uparrow \Rightarrow^{-} t \mid \Delta}{\Gamma;\varnothing \vdash \Box_{r} A \Downarrow \Rightarrow^{-} t \mid \Gamma - r \cdot (\Gamma - \Delta)}\;\Box^{-}_{R}$$

$$\dfrac{}{\Gamma \vdash 1 \Rightarrow^{-} () \mid \Gamma}\;1^{-}_{R} \quad \dfrac{\Gamma;\varnothing \vdash A \Uparrow \Rightarrow^{-} t \mid \Delta}{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow^{-} t \mid \Delta}\;\Downarrow^{-}_{R}$$

Figure 3.7: Right Sync rules of the focused subtractive synthesis calculus

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

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

$$\dfrac{\Gamma; x : A \Uparrow \vdash C \Rightarrow^{-} t \mid \Delta \qquad \text{A not atomic and not left sync}}{\Gamma; x : A \Downarrow \vdash C \Rightarrow^{-} t \mid \Delta}\;\Downarrow^{-}_{L}$$

Figure 3.8: Left Sync and Var rules of the focused subtractive synthesis calculus

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

$$\dfrac{\Gamma; \Omega \Uparrow \vdash C \Rightarrow t \mid \Delta \qquad C \text{ not right async}}{\Gamma; \Omega \vdash C \Uparrow \Rightarrow t \mid \Delta}\;\Uparrow^{+}_{R}$$

Figure 3.9: Right Async rules of the focused additive synthesis calculus

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

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

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

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

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

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

Figure 3.10: Left Async rules of the focused additive synthesis calculus

$$\frac{\Gamma; \emptyset \vdash C \Downarrow \Rightarrow t \mid \Delta \qquad \text{C not atomic}}{\Gamma; \emptyset \Uparrow \vdash C \Rightarrow t \mid \Delta}\ \textsc{focus}_R^+$$

$$\frac{\Gamma; x : A \Downarrow \vdash C \Rightarrow t \mid \Delta}{\Gamma, x : A; \emptyset \Uparrow \vdash C \Rightarrow t \mid \Delta}\ \textsc{focus}_L^+$$

Figure 3.11: Focus rules of the focused additive synthesis calculus

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

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

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

$$\dfrac{\Gamma;\varnothing \vdash A \Uparrow \Rightarrow t \mid \Delta}{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta}\ {\Downarrow}_R^+$$

Figure 3.12: Right Sync rules of the focused additive synthesis calculus

$$\textsc{LeftSync}$$

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

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

$$\dfrac{\Gamma; x : A \Uparrow \vdash C \Rightarrow t \mid \Delta \qquad \text{A not atomic and not left sync}}{\Gamma; x : A \Downarrow \vdash C \Rightarrow t \mid \Delta}\ {\Downarrow}_L^+$$

Figure 3.13: Left Sync and Var rules of the focused additive synthesis calculus

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

$$\dfrac{\Gamma;\varnothing \vdash A \Rightarrow t_1 \mid \Delta_1 \qquad \Gamma - \Delta_1;\varnothing \vdash B \Rightarrow t_2 \mid \Delta_2}{\Gamma;\varnothing \vdash A \otimes B \Rightarrow (t_1, t_2) \mid \Delta_1 + \Delta_2}\ {\otimes}_R^{\prime +}$$

Figure 3.14: Rules of the focused additive pruning synthesis calculus

One way to view focusing is in terms of a finite state machine, such as figure 3.15. 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, as well as the size of Ω, to decide whether to transition between focusing phases.

This focused approach to synthesis ensures that we are restricted to generating programs in β-normal form, which eliminates a class of redundant programs for which behaviourally equivalent β-normal forms can be synthesised in less steps.

Lemma 3.4.1 (Soundness of focusing). *For all contexts* Γ, Ω *and types* A*:*

$$\begin{array}{llcl}
1.\ \textit{Right Async}: & \Gamma;\Omega \vdash A \Uparrow\Rightarrow t \mid \Delta & \Longrightarrow & \Gamma, ,\Omega \vdash A \Rightarrow t \mid \Delta \\
2.\ \textit{Left Async}: & \Gamma;\Omega \Uparrow\vdash B \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma, ,\Omega \vdash B \Rightarrow t \mid \Delta \\
3.\ \textit{Right Sync}: & \Gamma;\emptyset \vdash A \Downarrow\ \Rightarrow t \mid\ \Delta & \Longrightarrow & \Gamma \vdash A \Rightarrow t \mid \Delta \\
4.\ \textit{Left Sync}: & \Gamma;x : A \Downarrow\vdash B \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma, x : A \vdash B \Rightarrow t \mid \Delta \\
5.\ \textit{Focus Right}: & \Gamma;\emptyset \vdash B \Rightarrow t \mid\ \Delta & \Longrightarrow & \Gamma \vdash B \Rightarrow t \mid \Delta \\
6.\ \textit{Focus Left}: & \Gamma, x : A;\emptyset \vdash B \Rightarrow t \mid\ \Delta & \Longrightarrow & \Gamma, x : A \vdash B \Rightarrow t \mid \Delta
\end{array}$$

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

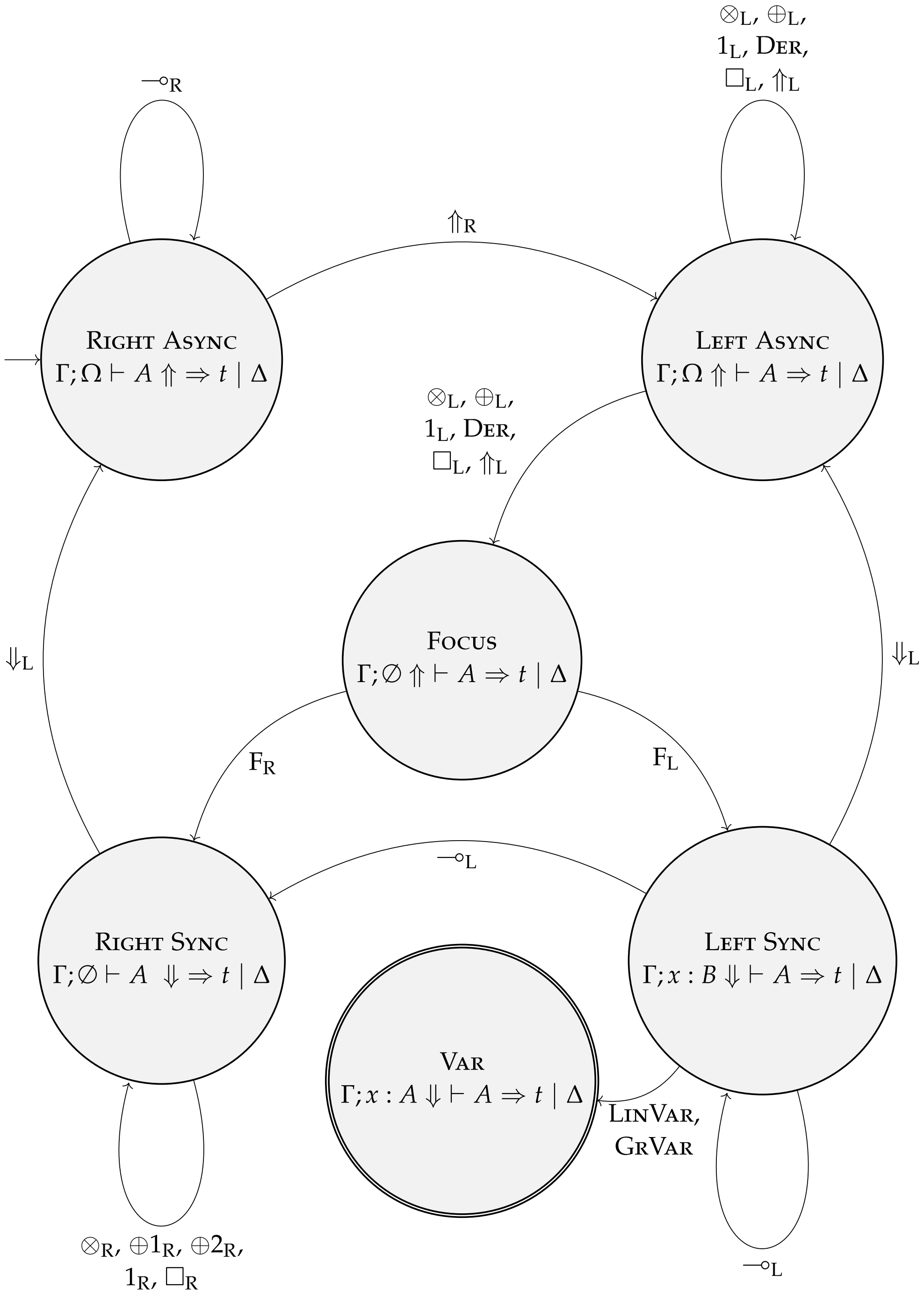

Figure 3.15: Focusing State Machine

3.5 EVALUATING THE RESOURCE MANAGEMENT SCHEMES

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

H1. (**Solving; Additive requires less**) 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);

H2. (**Paths; Subtractive explores fewer**) 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.

H3. (**Performance; additive faster on simpler examples**) 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.

3.5.1 *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. Multiplicative products are primitive in Granule, although additives coproducts are provided via ADTs, from which we define a core sum type to use here.

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 [41]. We use the LogicT monad for backtracking search [33] and the Scrap Your Reprinter library for splicing synthesised code into syntactic "holes", preserving the rest of the program text [15].

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 2.3.1 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 ! [24], I/O using ! [35], 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, e.g., $\forall r \in \mathbb{N}$, or $\forall r \in \mathcal{R}$ in any semiring, or $r = 0...\infty$:

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

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

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

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

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

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

Figure **??** provides the complete list of type schemes for these synthesis problems (32 in total). Note that $\Box A$ is used as shorthand for $\Box_{0...\infty} A$ (graded modality with indices drawn from intervals over $\mathbb{N} \cup \infty$)

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.

3.5.2 *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 3.1 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.

Hilbert	
⊗Intro	$\otimes_i : \forall a, b.\ a \multimap b \multimap (a \otimes b)$
⊗Elim	$\otimes_{e1} : \forall a, b.\ (a \otimes \Box_0 b) \multimap a$
	$\otimes_{e2} : \forall a, b.\ ()(\Box_0 a \otimes b) \multimap b$
⊕Intro	$\oplus_{i1} : \forall a, b.\ a \multimap a \oplus b$
	$\oplus_{i2} : \forall a, b.\ b \multimap a \oplus b$
⊕Elim	$\oplus_e : \forall a, b, c.\ \Box_{0...1}(a \multimap c) \multimap \Box_{0...1}(b \multimap c) \multimap (a \oplus b) \multimap c$
SKI	$s : \forall a, b, c.\ (a \multimap (b \multimap c)) \multimap (a \multimap b) \multimap (\Box_2 a \multimap c)$
	$k : \forall a, b.\ a \multimap \Box_0 b \multimap a$
	$i : \forall a.\ a \multimap a$
Comp	
mult	$\circ : \forall a, b, c.\ (a \multimap b) \multimap (b \multimap c) \multimap (a \multimap c)$
0/1	$\circ_{I/O} : \forall a, b, c.\ \Box(\Box a \multimap \Box b) \multimap \Box(\Box b \multimap \Box c) \multimap \Box(\Box a \multimap c)$
CBN	$\circ_{\text{CBN}} : \forall a, b, c.\ \Box(\Box a \multimap b) \multimap \Box(\Box b \multimap c) \multimap \Box a \multimap c$
CBV	$\circ_{\text{CBV}} : \forall a, b, c.\ \Box(\Box a \multimap \Box b) \multimap \Box(\Box b \multimap \Box c) \multimap \Box\Box a \multimap \Box c$
coK-$\mathcal{R}$	$\circ_{\mathcal{R}} : \forall \mathcal{R}, r, s \in \mathcal{R}, a, b, c.\ \Box_r(\Box_s a \multimap b) \multimap (\Box_r b \multimap c) \multimap \Box_{r \cdot s} a \multimap c$
coK-$\mathbb{N}$	$\circ_{\mathbb{N}} : \forall r, s \in \mathbb{N}, a, b, c.\ \Box_r(\Box_s a \multimap b) \multimap (\Box_r b \multimap c) \multimap \Box_{r \cdot s} a \multimap c$
Dist	
⊕-$\mathbb{N}$	$pull_{\oplus} : \forall r : \mathbb{N}, a, b.\ (\Box_r a \oplus \Box_r b) \multimap \Box_r(a \oplus b)$
⊕-!	$pull_{\oplus} : \forall a, b.\ (\Box a \oplus \Box b) \multimap \Box(a \oplus b)$
⊕-$\mathcal{R}$	$pull_{\oplus} : \forall \mathcal{R}, r \in \mathcal{R}, a, b.\ (\Box_r a \oplus \Box_r b) \multimap \Box_r(a \oplus b)$
⊗-$\mathbb{N}$	$pull_{\otimes} : \forall r : \mathbb{N}, a, b.\ (\Box_r a \otimes \Box_r b) \multimap \Box_r(a \otimes b)$
⊗-!	$pull_{\otimes} : \forall a, b.\ (\Box a \otimes \Box b) \multimap \Box(a \otimes b)$
⊗-$\mathbb{R}$	$pull_{\otimes} : \forall \mathcal{R}, r, a, b.\ (\Box_r a \otimes \Box_r b) \multimap \Box_r(a \otimes b)$
⊸-$\mathbb{N}$	$push_{\multimap} : \forall r : \mathbb{N}, a, b.\ \Box_r(a \multimap b) \multimap \Box_r a \multimap \Box_r b$
⊸-!	$push_{\multimap} : \forall a, b.\ \Box(a \multimap b) \multimap \Box a \multimap \Box b$
⊸-$\mathcal{R}$	$push_{\multimap} : \forall \mathcal{R}, r : \mathcal{R}, a, b.\ \Box_r(a \multimap b) \multimap \Box_r a \multimap \Box_r b$
Vec	
vec5	$vmap_5 : \forall a, b.\ \Box_5(a \multimap b) \multimap ((((a \otimes a) \otimes a) \otimes a) \otimes a)$ $\multimap ((((b \otimes b) \otimes b) \otimes b) \otimes b)$
vec10	$vmap_{10} : \forall a, b.$ *as above but for 10-tuples*
vec15	$vmap_{15} : \forall a, b.$ *as above but for 15-tuples*
vec20	$vmap_{20} : \forall a, b.$ *as above but for 20-tuples*
Misc	
split⊕	$split : \forall a, b, c. \Box_{2...3} b \multimap (a \oplus c) \multimap ((a \otimes \Box_{2..2} b) \oplus (c \otimes \Box_{3...3} b))$
split⊗	$split : \forall a, b. \Box_{0...2}(a \multimap a \multimap a) \multimap \Box_{10...10} a \multimap (\Box_{2...2} a \otimes \Box_{6...6} a)$
share	$share : \forall a, b. \Box_4 a \multimap \Box_6 a \multimap \Box_2(((((a \otimes a) \otimes a) \otimes a) \otimes a) \multimap b) \multimap (b \otimes b)$
Exm. 3.1.2	$noLeak : \forall a, b.(\Box_{\mathsf{Lo}} a \otimes \Box_{\mathsf{Hi}} a) \multimap (\Box_{\mathsf{Lo}}(a \otimes 1) \multimap b) \multimap b$

Figure 3.16: List of benchmark synthesis problems

3.5.2.1 *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 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.

Notably, on examples which are purely linear such as *andElim* from Hilbert's axioms or *mult* for function composition, the subtractive approach generally performs better. Linear programs without graded modalities can be synthesised without the need to interface with Z3 at all, making the differences here somewhat negligible as solver time generally makes up for the vast proportion of total synthesis time.

3.5.2.2 *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

At this point we have constructed a simple program synthesis tool for Granule, paramterised by a resource management scheme, which effectively deals with the problems of treating data as a resource inside a program. Both schemes would be a reasonable choice for further development of a synthesis tool for our language based on the graded linear λ-calculus.

Going forward, however, we focus primarily on the additive resource management scheme, using this as the basis for our more feature-rich fully-graded synthesis calculus in chapter 5. The evaluation in section 3.5 showed that the additive approach generally yields smaller and simpler theorems than the subtractive, requiring less time to solve. Theorem proving becomes even more prevalent in synthesis for a fully graded typing calculus - potentially every rule introduces new constraints that require solving, thus the speed at which this can be carried out is especially important.

While the tool presented in this chapter allows users to synthesise a considerable subset of Granule programs, it is still fairly 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. One notable limitation of our calculi is the inability to synthesise programs which perform a deep pattern match over a graded data type. A clear example of this can be found in the synthesis of programs which distribute a graded modality over a data type. Consider a classic example of a 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?

We instatiate the $\multimap_R^+$ rule at this type, building a partial synthesis derivation. Although we use the additive scheme for this example, the exact same situation arises in the subtractive.

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

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:

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

Clearly, this cannot 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 derivable in our calculi.

In the following chapter, we present an alternative approach to generating programs which exhibit this distributive behaviour using a generic programming methodology. The approach we present in chapter 4 is not type-directed program synthesis, per se. This approach complements the calculi presented here and in 5, providing users with a means to automatically generate programs purely from a type for a common class of graded programs. In describing this mechanism, we also begin to enhance our language with more advanced features such pattern matching, and recursion, further laying the foundations for chapter 5.

		Additive			Additive (pruning)			Subtractive		
	Problem		μT (ms)	N		μT (ms)	N		μT (ms)	N
Hilbert	⊗Intro	✓	6.69 (0.05)	2	✓	9.66 (0.23)	2	✓	10.93 (0.31)	2
	⊗Elim	✓	0.22 (0.01)	0	✓	0.05 (0.00)	0	✓	0.06 (0.00)	0
	⊕Intro	✓	0.08 (0.00)	0	✓	0.07 (0.00)	0	✓	0.07 (0.00)	0
	⊕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	⊗-!	✓	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
	⊕-!	✓	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
	⊸-!	✓	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⊕	✓	3.79 (0.04)	1	✓	5.10 (0.16)	1	✓	10732.65 (8.01)	6
	split⊗	✓	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 3.1: Results. μT in *ms* to 2 d.p. with standard sample error in brackets

4

AUTOMATICALLY 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 $\square_r(\mathsf{F}\alpha)$ 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 $\mathsf{F}(\square_r\alpha)$ value, or perhaps vice versa. A *distributive law* (in the categorical sense, e.g., [49]) provides a conversion from one to the other. In this chapter, we present a procedure to automatically synthesise these distributive operators, applying a generic programming methodology [28] 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.

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

As a standalone methodology for generating a common class of graded programs, this "deriving mechanism" serves as a complement to the synthesis calculi of chapter 3. Synthesis problems which exhibit this distributive behaviour pose issues in the existing calculi. However, in many cases the solution programs to these distributive problems are straightforwardly derivable from the type alone, making the costly enumerative search of type-directed synthesis unnecessary.

Thus, we present a tool for an automatic procedure which calculates distributive laws from graded types and present a formal analysis of their properties. This approach is realised in Granule, embedded into the compiler. In doing so, we extend our graded linear λ-calculus of section 2.3 to incorporate data constructors and pattern matching, as well as recursive data types.

This extended calculus is defined in section **??** providing an idealised, simply-typed subset of Granule with which we develop the core deriving mechanism. Section 4.2 gives the procedures for deriving *push* and *pull* operators for the calculus. Section **??** describes the details of how these procedures are realised in the Granule language. We then provide examples of how several other structural combinators in Granule may be derivinged using this tool in section **??**. Finally, section **??** discusses more related and future work.

We start with a motivating example typifying the kind of software engineering impedance problem that distributive laws solve. We do so in Granule code since it is the main vehicle for the developments here.

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. 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 $\Box_0 b$.

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
```

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 chapter 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* [28]. 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 ⊓ m]
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, $\sqcap$ 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 GRADED LINEAR-λ-CALCULUS

We define here a typing calculus which extends the graded linear λ-calculus of 2.3 with data constructors, pattern matching, and recursive data types. This language constitutes a simplified monomorphic subset of Granule. We include notions of data constructors and their elimination via **case** expressions as a way to unify the handling of regular type constructors.

The full syntax of terms and types is given by:

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

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

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

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

4.1.1 *Typing*

For the most part, typing follows the calculus defined in section 3.1. Figure **??** gives the complete typing rules. We briefly explain the extensions introduced for this chapter.

The LETREC rule provides recursive bindings in the standard way.

Data constructors with zero or more arguments are introduced via the CON rule. Here, the constructors that concern us are units,

$$\dfrac{}{x : A \vdash x : A}\textsc{Var} \qquad \dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\textsc{Abs}$$

$$\dfrac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\textsc{App} \qquad \dfrac{[\Gamma] \vdash t : A}{r \cdot [\Gamma] \vdash [t] : \Box_r A}\textsc{Pr}$$

$$\dfrac{\Gamma, x : A \vdash t : B}{\Gamma, x :_1 A \vdash t : B}\textsc{Der} \qquad \dfrac{\Gamma \vdash t : A}{\Gamma, [\Delta]_0 \vdash t : A}\textsc{weak}$$

$$\dfrac{\Gamma, x :_r A, \Gamma' \vdash t : A \qquad r \sqsubseteq s}{\Gamma, x :_s A, \Gamma' \vdash t : A}\textsc{Approx}$$

$$\dfrac{\Gamma, x : A \vdash t_1 : A \qquad \Gamma', x : A \vdash t_2 : B}{\Gamma + \Gamma' \vdash \textbf{letrec}\ x \ = \ t_1\ \textbf{in}\ t_2 : B}\textsc{Letrec}$$

$$\dfrac{(C : B_1 \multimap ... \multimap B_n \multimap A) \in \mathrm{D}}{\emptyset \ \vdash \ C : B_1 \multimap ... \multimap B_n \multimap A}\textsc{Con}$$

$$\dfrac{\Gamma \vdash t : A \quad \cdot \vdash p_i : A \rhd \Delta_i \quad \Gamma', \Delta_i \vdash t_i : B}{\Gamma + \Gamma' \vdash \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; ...; p_n \mapsto t_n : B}\textsc{Case}$$

Figure 4.1: Typing rules for the extended graded linear λ-calculus

products, and coproducts (sums), given by D, a global set of data constructors with their types, defined:

$$\begin{aligned} D = &\ \{() : 1\} \\ &\cup \{(,) : A \multimap B \multimap A \otimes B \mid \forall A, B\} \\ &\cup \{\mathsf{inl} : A \multimap A \oplus B \mid \forall A, B\} \\ &\cup \{\mathsf{inr} : B \multimap A \oplus B \mid \forall A, B\} \end{aligned}$$

Constructors are eliminated by pattern matching via the CASE rule. Patterns p are typed by judgments of the form $?r \vdash p : A \rhd \Delta$ meaning that a pattern p has type A and produces a context of typed binders Δ (used, e.g., in the typing of the case branches). The information to the left of the turnstile denotes optional grade information arising from being in an unboxing pattern and is syntactically defined as either:

$$r : ?R \ ::= - \mid r \qquad \text{(enclosing grade)}$$

where $-$ means the present pattern is not nested inside an unboxing pattern and r that the present pattern is nested inside an unboxing pattern for a graded modality with grade r.

$$\frac{}{\cdot \vdash x : A \rhd x : A}\ \textsc{PVar} \qquad \frac{\cdot \vdash p_i : B_i \rhd \Gamma_i}{\cdot \vdash C p_1 .. p_n : A \rhd \Gamma_1, .., \Gamma_n}\ \textsc{PCon}$$

$$\frac{r \vdash p : A \rhd \Gamma}{\cdot \vdash [p] : \Box_r A \rhd \Gamma}\ \textsc{PBox}$$

$$\frac{r \vdash p_i : B_i \rhd \Gamma_i \qquad |A| > 1 \Rightarrow 1 \sqsubseteq r}{r \vdash C p_1 .. p_n : A \rhd \Gamma_1, .., \Gamma_n}\ [\textsc{PCon}]$$

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

Figure 4.2: Pattern typing rules for the extended graded linear λ-calculus

The rules of pattern typing are given in Figure 4.2. The rule (PBox) provides graded modal elimination (an 'unboxing' pattern), propagating grade information into the typing of the sub-pattern. Thus **case** t **of** $[p] \to t'$ can be used to eliminate a graded modal value. Variable patterns are typed via two rules depending on whether the variable occurs inside an unbox pattern ([PVar]) or not (PVar), with the [PVar] rule producing a binding with the grade of the enclosing box's grade r. As with variable patterns, constructor patterns are split between rules for patterns which either occur inside an unboxing pattern or not. In the former case, the grade information is propagated to the subpattern(s), with the additional constraint that if there is more than one data constructor for the type A (written $|A| > 1$), then the grade r must approximate 1 (written $1 \sqsubseteq r$) as pattern matching incurs a usage to inspect the constructor. The operation $|A|$ counts the number of data constructors for a type:

$$|1| = 1 \quad |A \multimap B| = 1 \quad |\Box_r A| = |A|$$
$$|A \oplus B| = 2(|A| + |B|) \quad |A \otimes B| = |A||B|$$
$$|\mu X.A| = |A[\mu X.A/X]|$$

and $|X|$ is undefined (or effectively 0) since we do not allow unguarded recursion variables in types. A type A must therefore involve a sum type for $|A| > 1$.

Since a wildcard pattern _ discards a value, this is only allowed inside an unboxing pattern where the enclosing grade permits weakening, captured via $0 \sqsubseteq r$ in rule [Pwild].

4.2 AUTOMATICALLY DERIVING *push* AND *pull*

Now that we have established the language, we describe the algorithmic calculation of distributive laws. Note that whilst our language 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.

4.2.1 *Notation*

Let $\mathsf{F} : \mathsf{Type}^n \to \mathsf{Type}$ 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 $1 \leq i \leq n$.

4.2.2 *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 $[\![A]\!]_{\mathsf{push}} = \lambda z.[\![A]\!]^{\varnothing}_{\mathsf{push}}\ z$ given by an intermediate interpretation $[\![A]\!]^{\Sigma}_{\mathsf{push}}$ where Σ is a context of *push* combinators for the recursive type variables. This interpretation is defined by Figure 4.3. In the case of *push* on a value of type 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

$$\begin{aligned}
[\![1]\!]^{\Sigma}_{\mathsf{push}}\ z &= \textbf{case}\ z\ \textbf{of}\ [()] \rightarrow () \\
[\![\alpha]\!]^{\Sigma}_{\mathsf{push}}\ z &= z \\
[\![X]\!]^{\Sigma}_{\mathsf{push}}\ z &= \Sigma(X)\ z \\
[\![A \oplus B]\!]^{\Sigma}_{\mathsf{push}}\ z &= \textbf{case}\ z\ \textbf{of}\ \ [\mathsf{inl}\ x] \rightarrow \mathsf{inl}\ [\![A]\!]^{\Sigma}_{\mathsf{push}}[x]; \\
&\qquad\qquad\qquad [\mathsf{inr}\ y] \rightarrow \mathsf{inr}\ [\![B]\!]^{\Sigma}_{\mathsf{push}}[y] \\
[\![A \otimes B]\!]^{\Sigma}_{\mathsf{push}}\ z &= \textbf{case}\ z\ \textbf{of}\ [(x,y)] \rightarrow ([\![A]\!]^{\Sigma}_{\mathsf{push}}[x], [\![B]\!]^{\Sigma}_{\mathsf{push}}[y]) \\
[\![A \multimap B]\!]^{\Sigma}_{\mathsf{push}}\ z &= \lambda y.\textbf{case}\ z\ \textbf{of}\ [f] \rightarrow \\
&\qquad\qquad \textbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{pull}}\ y\ \textbf{of}\ [u] \rightarrow [\![B]\!]^{\Sigma}_{\mathsf{push}}[(f\ u)] \\
[\![\mu X.A]\!]^{\Sigma}_{\mathsf{push}}\ z &= \textbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \mu X.\Box_r A \multimap (\mu X.A)\overrightarrow{[\Box_r \alpha_i / \alpha_i]}}_{\mathsf{push}}\ \textbf{in}\ f\ z
\end{aligned}$$

Figure 4.3: Interpretation rules for $[\![A]\!]_{\mathsf{push}}$

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 **??**).

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

4.2.3 *Pull*

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

$$[\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{pull}} : \mathsf{F}\ (\overline{\Box_{r_i}\alpha_i}) \multimap \Box_{\bigwedge_{i=1}^{n} r_i}(\mathsf{F}\ \overline{\alpha_i})$$

Type constructor F here is applied to n arguments each of the form $\Box_{r_i}\alpha_i$, i.e., each with a different grading of which the greatest-lower bound[1] $\bigwedge_{i=1}^{n} r_i$ is the resulting grade (see `pullPair` from Section 4.0.1).

For types A closed with respect to recursion variables, let $[\![A]\!]_{\mathsf{pull}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{pull}}\ z$ given by an intermediate interpretation

[1] The greatest-lower bound $\wedge$ is partial operation which can be defined in terms of the semiring's pre-order: $r \wedge s = t$ if $t \sqsubseteq r$, $t \sqsubseteq s$ and there exists no other t' where $t' \sqsubseteq r$ and $t' \sqsubseteq s$ and $t \sqsubseteq t'$.

$$
\begin{aligned}
[\![1]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \textbf{case}\ z\ \textbf{of}\ () \rightarrow [()] \\
[\![\alpha]\!]^{\Sigma}_{\mathsf{pull}}\ z &= z \\
[\![X]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \Sigma(X)\ z \\
[\![A \oplus B]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \textbf{case}\ z\ \textbf{of}\ \ \begin{array}{l}\mathsf{inl}\ x \rightarrow \textbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{pull}}\ x\ \textbf{of}\ [u] \rightarrow [\mathsf{inl}\ u]; \\ \mathsf{inr}\ y \rightarrow \textbf{case}\ [\![B]\!]^{\Sigma}_{\mathsf{pull}}\ y\ \textbf{of}\ [v] \rightarrow [\mathsf{inr}\ v]\end{array} \\
[\![A \otimes B]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \textbf{case}\ z\ \textbf{of}\ (x, y) \rightarrow \\
&\qquad\qquad \textbf{case}\ ([\![A]\!]^{\Sigma}_{\mathsf{pull}}\ x, [\![B]\!]^{\Sigma}_{\mathsf{pull}}\ y)\ \textbf{of}\ ([u], [v]) \rightarrow [(u, v)] \\
[\![\mu X.A]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \textbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \mu X.A\overrightarrow{[\Box_{r_i}\alpha_i/\alpha_i]} \multimap \Box_{\bigwedge_{i=1}^{n} r_i}(\mu X.A)}_{\mathsf{pull}}\ \ \textbf{in}\ f\ z
\end{aligned}
$$

Figure 4.4: Interpretation rules for $[\![A]\!]_{\mathsf{pull}}$

$[\![A]\!]^{\Sigma}_{\mathsf{pull}}$ where Σ is a context of *pull* combinators for the recursive type variables. This interpretation is defined by Figure ??.

Just like *push*, we cannot apply *pull* to graded modalities themselves. Unlike *push*, we cannot apply *pull* to function types. That is, we cannot derive a distributive law of the form $(\Box_r A \multimap \Box_r B) \multimap \Box_r (A \multimap B)$ since introducing the concluding $\Box_r$ would require the incoming function $(\Box_r A \multimap \Box_r B)$ to itself be inside $\Box_r$ due to the promotion rule (PR), which does not match the type scheme for *pull*.

The rest of the derivation above is similar but dual to that of *push*.

Appendix **??** gives the proof that $[\![A]\!]_{\mathsf{pull}}$ is type sound, i.e., its derivations are well-typed.

Example 4.2.1. To illustrate the above procedures, the derivation of $\lambda z.[\![(\alpha \otimes \alpha) \multimap \beta]\!]_{\mathsf{push}}\ z : \Box_r((\alpha \otimes \alpha) \multimap \beta) \multimap ((\Box_r \alpha \otimes \Box_r \alpha) \multimap \Box_r \beta)$ is:

$$
\begin{aligned}
& \lambda z.[\![(\alpha \otimes \alpha) \multimap \beta]\!]^{\emptyset}_{\mathsf{push}}\ z \\
= {} & \lambda z.\lambda y.\mathbf{case}\ z\ \mathbf{of}\ [f] \to \mathbf{case}\ [\![\alpha \otimes \alpha]\!]^{\emptyset}_{\mathsf{pull}}\ y\ \mathbf{of}\ [u] \to [\![\beta]\!]^{\emptyset}_{\mathsf{push}}[(f\ u)] \\
= {} & \lambda z.\lambda y.\mathbf{case}\ z\ \mathbf{of}\ [f] \to \\
& \qquad \mathbf{case}\ (\mathbf{case}\ y\ \mathbf{of}\ (x', y') \to \\
& \qquad\qquad \mathbf{case}\ ([\![\alpha]\!]^{\emptyset}_{\mathsf{pull}}\ x', [\![\alpha]\!]^{\emptyset}_{\mathsf{pull}}\ y')\ \mathbf{of}\ ([u],[v]) \to [(u,v)])\ \mathbf{of} \\
& \qquad\quad [u] \to [\![\beta]\!]^{\emptyset}_{\mathsf{push}}[(f\ u)] \\
= {} & \lambda z.\lambda y.\mathbf{case}\ z\ \mathbf{of}\ [f] \to \\
& \qquad \mathbf{case}\ (\mathbf{case}\ y\ \mathbf{of}\ (x', y') \to \\
& \qquad\qquad \mathbf{case}\ (x', y')\ \mathbf{of}\ ([u],[v]) \to [(u,v)])\ \mathbf{of}\ [u] \to [(f\ u)]
\end{aligned}
$$

Remark 1. One might ponder whether linear logic's exponential $!A$ [24] is modelled by the graded necessity modality over $\mathbb{N}_\infty$ intervals, i.e., with $!A \triangleq \Box_{0..\infty} A$. This is a reasonable assumption, but $\Box_{0..\infty} A$ has a slightly different meaning to $!A$, exposed here: whilst $[\![A \otimes B]\!]_{\mathsf{push}} : \Box_{0..\infty}(A \otimes B) \multimap (\Box_{0..\infty} A \otimes \Box_{0..\infty} B)$ is derivable in our language, linear logic does not permit $!(A \otimes B) \multimap (!A \otimes !B)$. Models of ! provide only a monoidal functor structure which gives *pull* for $\otimes$, but not *push* [6]. This structure can be recovered in Granule through the introduction of a partial type-level operation which selectively disallows *push* for $\otimes$ in semirings which model the ! modality of linear logic[2] [30].

The algorithmic definitions of 'push' and 'pull' can be leveraged in a programming context to automatically yield these combinators for practical purposes. We discuss how this is leveraged inside the Granule compiler in Section **??**. Before that, we study the algebraic behaviour of the derived distributive laws.

4.2.4 *Properties*

We consider here the properties of these derived operations. Prima facie, the above *push* and *pull* operations are simply distributive laws between two (parametric) type constructors F and $\Box_r$, the latter being the graded modality. However, both F and $\Box_r$ have additional structure. If the mathematical terminology of 'distributive laws' is warranted, then such additional structure

[2] The work in [30] arose as a result of this work.

should be preserved by *push* and *pull* (e.g., as in how a distributive law between a monad and a comonad must preserve the behaviour of the monad and comonad operations after applying the distributive law [47]); we explain here the relevant additional structure and verify the distributive law properties.

We note that these distributive laws are mutually inverse:

Proposition 4.2.1 (Pull is right inverse to push)**.** *For all n-arity types* F *which do not contain function types, then for type variables* $(\alpha_i)_{i\in[1..n]}$ *and for all grades* $r \in \mathcal{R}$ *where* $1 \sqsubseteq r$ *if* $|\mathsf{F}\overline{\alpha_i}| > 1$*, then:*

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

Proposition 4.2.2 (Pull is left inverse to push)**.** *For all n-arity types* F *which do not contain function types, then for type variables* $(\alpha_i)_{i\in[1..n]}$ *and for all grades* $r \in \mathcal{R}$ *where* $1 \sqsubseteq r$ *if* $|\mathsf{F}\overline{\alpha_i}| > 1$*, then:*

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

The appendix **??** gives the proofs, leveraging an equational theory for our lanugage.

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 our language 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 [17]) 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 (a [r])
```

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 re-calculated 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.

4.3.1 *Examples*

At the start of this chapter, we motivated the crux of this work 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))
-- where, copy : a [2] → (a, a)
copyPair x = copy (pull @(,) x)
```

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 + 1) a) [0..1] → a
```

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)
-- [y] unboxes (a [2]) to y:a usable twice
copyHead' xs = let [y] = head xs in (y, y)
```

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] → b [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 (a [r])
```

4.4 DERIVING OTHER USEFUL STRUCTURAL 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.

4.4.1 *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 derivation procedure of ?? to derive weakening on compound types.

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}.

4.4.2 *A Combinator for Copying "shape"*

The "shape" of values for a parametric data types F can be determined by a function $\mathit{shape} : \mathsf{F}A \rightarrow \mathsf{F}1$, usually derived when F is a functor by mapping with $A \rightarrow 1$ (dropping elements) [31].

$$
\begin{aligned}
\llbracket C^w \rrbracket^{\Sigma}_{\mathsf{drop}} z &= \mathtt{drop}\ z \\
\llbracket 1 \rrbracket^{\Sigma}_{\mathsf{drop}} z &= \mathbf{case}\ z\ \mathbf{of}\ () \to () \\
\llbracket X \rrbracket^{\Sigma}_{\mathsf{drop}} z &= \Sigma(X) z \\
\llbracket A \oplus B \rrbracket^{\Sigma}_{\mathsf{drop}} z &= \mathbf{case}\ z\ \mathbf{of}\ \ \mathsf{inl}\ x \to \llbracket A \rrbracket_{\mathsf{drop}}(x);\ \ \mathsf{inr}\ y \to \llbracket B \rrbracket_{\mathsf{drop}}(y) \\
\llbracket A \otimes B \rrbracket^{\Sigma}_{\mathsf{drop}} z &= \mathbf{case}\ z\ \mathbf{of}\ (x, y) \to \\
&\qquad\qquad \mathbf{case}\ \llbracket A \rrbracket_{\mathsf{drop}}(x)\ \mathbf{of}\ () \to \\
&\qquad\qquad\qquad \mathbf{case}\ \llbracket B \rrbracket_{\mathsf{drop}}(y)\ \mathbf{of}\ () \to () \\
\llbracket \mu X.A \rrbracket^{\Sigma}_{\mathsf{drop}} z &= \mathbf{letrec}\ f = \llbracket A \rrbracket^{\Sigma, X \mapsto f : A \multimap 1}_{\mathsf{drop}}\ \mathbf{in}\ f\ z
\end{aligned}
$$

Figure 4.5: Interpretation rules for $\llbracket A \rrbracket_{\mathsf{drop}}$

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:

$$\llbracket \mathsf{F}\alpha \rrbracket_{\mathsf{copyShape}} : \mathsf{F}\alpha \multimap \mathsf{F}1 \otimes \mathsf{F}\alpha$$

defined by $\llbracket A \rrbracket_{\mathsf{copyShape}} = \lambda z. \llbracket A \rrbracket^{\emptyset}_{\mathsf{copyShape}} z$ by an intermediate interpretation $\llbracket A \rrbracket^{\Sigma}_{\mathsf{copyShape}}$, given by Figure 4.6. The implementation recursively follows the structure of the type, replicating the

$$
\begin{aligned}
\llbracket C^{w} \rrbracket^{\Sigma}_{\mathsf{copyShape}} z &= ((),\ z)\\
\llbracket 1 \rrbracket^{\Sigma}_{\mathsf{copyShape}} z &= \textbf{case}\ z\ \textbf{of}\ () \rightarrow ((),\ ())\\
\llbracket \alpha \rrbracket^{\Sigma}_{\mathsf{copyShape}} z &= ((), z)\\
\llbracket X \rrbracket^{\Sigma}_{\mathsf{copyShape}} z &= \Sigma(X) z\\
\llbracket A \oplus B \rrbracket^{\Sigma}_{\mathsf{copyShape}} z &= \textbf{case}\ z\ \textbf{of}\\
&\qquad\qquad \mathsf{inl}\ x \rightarrow \textbf{case}\ \llbracket A \rrbracket^{\Sigma}_{\mathsf{copyShape}}(x)\ \textbf{of}\ (s,\ x') \rightarrow\\
&\qquad\qquad\qquad\qquad (\mathsf{inl}\ s,\ \mathsf{inr}\ x')\\
&\qquad\qquad \mathsf{inr}\ y \rightarrow \textbf{case}\ \llbracket B \rrbracket^{\Sigma}_{\mathsf{copyShape}}(y)\ \textbf{of}\ (s,\ y') \rightarrow\\
&\qquad\qquad\qquad\qquad (\mathsf{inr}\ s,\ \mathsf{inr}\ y')\\
\llbracket A \otimes B \rrbracket^{\Sigma}_{\mathsf{copyShape}} z &= \textbf{case}\ z\ \textbf{of}\ (x, y) \rightarrow\\
&\qquad\qquad \textbf{case}\ \llbracket A \rrbracket^{\Sigma}_{\mathsf{copyShape}}(x)\ \textbf{of}\ (s,\ x') \rightarrow\\
&\qquad\qquad\qquad \textbf{case}\ \llbracket B \rrbracket^{\Sigma}_{\mathsf{copyShape}}(y)\ \textbf{of}\\
&\qquad\qquad\qquad\qquad (s',\ y') \rightarrow ((s,\ s'),\ (x',\ y'))\\
\llbracket \mu X.A \rrbracket^{\Sigma}_{\mathsf{copyShape}} z &= \textbf{letrec}\ f = \llbracket A \rrbracket^{\Sigma, X \mapsto f : A \multimap 1 \otimes A}_{\mathsf{copyShape}}\ \textbf{in}\ f\ z
\end{aligned}
$$

Figure 4.6: Interpretation rules for $\llbracket A \rrbracket_{\mathsf{copyShape}}$

constructors, reaching the crucial case where a polymorphically type $z : \alpha$ is mapped to $((), z)$ 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`.

4.5 RELATED WORK

In this section we consider some of the wider related work as they relate to the ideas presented in this chapter.

4.5.1 *Generic Programming Methodology*

The deriving mechanism for Granule is based on the methodology of generic functional programming [28], 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 work is the work on generic deriving [37], which allows Haskell programmers to automatically derive arbitrary class instances using standard datatype-generic programming techniques as described above.

4.5.2 *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 [9]. Power and Watanabe study in detail different ways of combining comonads and monads via distributive laws [47]. Such distributive laws have been applied in the programming languages literature, e.g., for modelling streams of partial elements [50].

4.5.3 *Graded Distributive Laws*

Gaboardi et al. define families of graded distributive laws for graded monads and comonads [22]. 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. [22].

4.6 CONCLUSION

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.

5

AN EXTENDED SYNTHESIS CALCULUS

5.1 A FULLY GRADED TARGET LANGUAGE

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

$$\begin{aligned} t &::= x \mid \lambda x.t \mid t_1\, t_2 \mid [t] \mid C\, t_1 \ldots t_n \mid \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; \ldots; p_n \mapsto t_n \\ & && (\textit{terms}) \\ p &::= x \mid \_ \mid [p] \mid C\, p_1 \ldots p_n && (\textit{patterns}) \end{aligned}$$

$$\frac{\overline{\alpha : \kappa}; \emptyset \vdash t : A}{\emptyset; \emptyset \vdash t : \forall \overline{\alpha : \kappa}.A}\ \textsc{TopLevel}$$

5.2 A FULLY GRADED SYNTHESIS CALCULUS

5.3 FOCUSING THE FULLY GRADED SYNTHESIS CALCULUS

5.4 EVALUATION

5.5 CONCLUSION

$$\frac{\Sigma \vdash A : \mathrm{Type}}{\Sigma; 0 \cdot \Gamma, x :_1 A \vdash x : A}\ \textsc{Var}$$

$$\frac{(x : \forall \overline{\alpha : \kappa}.A') \in D \qquad \Sigma \vdash A = \mathrm{inst}(\forall \overline{\alpha : \kappa}.A')}{\Sigma; 0 \cdot \Gamma \vdash x : A}\ \textsc{Def}$$

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

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

$$\frac{\begin{array}{l}(C : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K \vec{A'}) \in D \\ \Sigma \vdash {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K \vec{A} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K \vec{A'})\end{array}}{\Sigma; 0 \cdot \Gamma \vdash C : {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K \vec{A}}\ \textsc{Con}$$

$$\frac{\Sigma; \Gamma \vdash t : A \qquad \Sigma; r \vdash p_i : A \rhd \Delta_i \qquad \Sigma; \Gamma', \Delta_i \vdash t_i : B}{\Sigma; r \cdot \Gamma + \Gamma' \vdash \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; \ldots; p_n \mapsto t_n : B}\ \textsc{Case}$$

$$\frac{\Sigma; \Gamma \vdash t : A[\mu X.A / X]}{\Sigma; \Gamma \vdash t : \mu X.A}\ \mu_1 \qquad \frac{\Sigma; \Gamma \vdash t : \mu X.A}{\Sigma; \Gamma \vdash t : A[\mu X.A / X]}\ \mu_2$$

Figure 5.1: Typing rules for fully graded typing calculus

$$\frac{0 \sqsubseteq r \qquad \Sigma \vdash A : \mathrm{Type}}{\Sigma; r \vdash \_ : A \rhd \emptyset}\ \textsc{PWild}$$

$$\frac{\Sigma \vdash A : \mathrm{Type}}{\Sigma; r \vdash x : A \rhd x :_r A}\ \textsc{PVar} \qquad \frac{\Sigma; r \cdot s \vdash p : A \rhd \Gamma}{\Sigma; r \vdash [p] : \Box_s A \rhd \Gamma}\ \textsc{PBox}$$

$$\frac{\begin{array}{l}(C : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K \vec{A'}) \in D \\ \Sigma \vdash {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K \vec{A} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K \vec{A'}) \\ \Sigma; q_i \cdot r \vdash p_i : B_i \rhd \Gamma_i \qquad |K \vec{A}| > 1 \Rightarrow 1 \sqsubseteq r\end{array}}{\Sigma; r \vdash C\, p_1 \ldots p_n : K \vec{A} \rhd \overrightarrow{\Gamma_i}}\ \textsc{PC}$$

Figure 5.2: Pattern typing rules of for the fully graded typing calculus

6

CONCLUSION

6.1 FUTURE DIRECTIONS

BIBLIOGRAPHY

[1] Andreas Abel and Jean-Philippe Bernardy. "A unified view of modalities in type systems." In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 90:1–90:28. DOI: 10.1145/3408972. URL: https://doi.org/10.1145/3408972.

[2] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. "Recursive Program Synthesis." In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 2013, pp. 934–950. DOI: 10.1007/978-3-642-39799-8_67. URL: https://doi.org/10.1007/978-3-642-39799-8_67.

[3] Guillaume Allais. "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. 2018.

[4] Jean-Marc Andreoli. "Logic Programming with Focusing Proofs in Linear Logic." In: *Journal of Logic and Computation* 2.3 (June 1992), pp. 297–347. ISSN: 0955-792X. DOI: 10.1093/logcom/2.3.297. eprint: https://academic.oup.com/logcom/article-pdf/2/3/297/6137548/2-3-297.pdf.

[5] Robert Atkey. "Syntax and Semantics of Quantitative Type Theory." In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. 2018, pp. 56–65. DOI: 10.1145/3209108.3209189. URL: https://doi.org/10.1145/3209108.3209189.

[6] Nick Benton, Gavin Bierman, Valeria De Paiva, and Martin Hyland. "Linear lambda-calculus and categorical models revisited." In: *International Workshop on Computer Science Logic*. Springer. 1992, pp. 61–84. DOI: 10.1007/3-540-56992-8_6.

[7] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. "Linear Haskell: practical linearity in a higher-order polymorphic

language." In: *Proc. ACM Program. Lang.* 2.POPL (2018), 5:1–5:29. DOI: 10.1145/3158093. URL: https://doi.org/10.1145/3158093.

[8] Edwin C. Brady. "Idris 2: Quantitative Type Theory in Practice." In: *CoRR* abs/2104.00480 (2021). arXiv: 2104.00480. URL: https://arxiv.org/abs/2104.00480.

[9] Stephen Brookes and Kathryn V Stone. *Monads and Comonads in Intensional Semantics*. Tech. rep. Pittsburgh, PA, USA, 1993. URL: https://dl.acm.org/doi/10.5555/865105.

[10] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. "A Core Quantitative Coeffect Calculus." In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 351–370. DOI: 10.1007/978-3-642-54833-8_19.

[11] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. "Efficient resource management for linear logic proof search." In: *Theoretical Computer Science* 232.1 (2000), pp. 133 –163. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(99)00173-5.

[12] Kaustuv Chaudhuri and Frank Pfenning. "A Focusing Inverse Method Theorem Prover for First-Order Linear Logic." In: *Proceedings of the 20th International Conference on Automated Deduction*. CADE' 20. Tallinn, Estonia: Springer-Verlag, 2005, 69–83. ISBN: 3540280057. DOI: 10.1007/11532231_6.

[13] Kaustuv Chaudhuri and Frank Pfenning. "Focusing the Inverse Method for Linear Logic." In: *Proceedings of the 19th International Conference on Computer Science Logic*. CSL'05. Oxford, UK: Springer-Verlag, 2005, 200–215. ISBN: 3540282319. DOI: 10.1007/11538363_15.

[14] Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. "A graded dependent type system with a usage-aware semantics." In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–32. DOI: 10.1145/3434331. URL: https://doi.org/10.1145/3434331.

[15] Harry Clarke, Vilem-Benjamin Liepelt, and Dominic Orchard. "Scrap Your Reprinter." In: (2017).

[16] Anatoli Degtyarev and Andrei Voronkov. "Chapter 4 - The Inverse Method." In: *Handbook of Automated Reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Handbook of Automated Reasoning. Amsterdam: North-Holland, 2001, pp. 179 –272. ISBN: 978-0-444-50813-3. DOI: https://doi.org/10.1016/B978-044450813-3/50006-0.

[17] Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. "Visible type application." In: *European Symposium on Programming*. Springer. 2016, pp. 229–254. DOI: 10.1007/978-3-662-49498-1_10.

[18] John K. Feser, Swarat Chaudhuri, and Isil Dillig. "Synthesizing Data Structure Transformations from Input-Output Examples." In: *SIGPLAN Not.* 50.6 (2015), 229–239. ISSN: 0362-1340. DOI: 10.1145/2813885.2737977. URL: https://doi.org/10.1145/2813885.2737977.

[19] Jonas Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. "Leveraging Rust Types for Program Synthesis." In: *To appear in the Proceedings of PLDI* (2023).

[20] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. "Example-directed synthesis: a type-theoretic interpretation." In: *ACM SIGPLAN Notices* 51.1 (2016), pp. 802–815.

[21] Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. "Combining effects and coeffects via grading." In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM, 2016, pp. 476–489. DOI: 10.1145/2951913.2951939.

[22] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. "Combining Effects and Coeffects via Grading." In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: Association for Computing Machinery, 2016, 476–489. ISBN: 9781450342193. DOI: 10.1145/2951913.2951939. URL: https://doi.org/10.1145/2951913.2951939.

[23] Dan R. Ghica and Alex I. Smith. “Bounded Linear Types in a Resource Semiring.” In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 331–350. DOI: 10.1007/978-3-642-54833-8_18.

[24] Jean-Yves Girard. “Linear logic.” In: *Theoretical Computer Science* 50.1 (1987), pp. 1 –101. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(87)90045-4.

[25] Jean-Yves Girard, Andre Scedrov, and Philip J Scott. “Bounded linear logic: a modular approach to polynomial-time computability.” In: *Theoretical computer science* 97.1 (1992), pp. 1–66. DOI: 10.1016/0304-3975(92)90386-T.

[26] Cordell Green. “Application of Theorem Proving to Problem Solving.” In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. IJCAI’69. Washington, DC: Morgan Kaufmann Publishers Inc., 1969, 219–239.

[27] James Harland and David J. Pym. “Resource-distribution via Boolean constraints.” In: *CoRR* cs.LO/0012018 (2000). URL: https://arxiv.org/abs/cs/0012018.

[28] Ralf Hinze. “A new approach to generic functional programming.” In: *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2000, pp. 119–132. DOI: 10.1145/325694.325709.

[29] J.S. Hodas and D. Miller. “Logic Programming in a Fragment of Intuitionistic Linear Logic.” In: *Information and Computation* 110.2 (1994), pp. 327 –365. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1036.

[30] Jack Hughes, Daniel Marshall, James Wood, and Dominic Orchard. “Linear Exponentials as Graded Modal Types.” In: *5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021)*. Rome (virtual), Italy, June 2021. URL: https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271465.

[31] C Barry Jay and J Robin B Cockett. “Shapely types and shape polymorphism.” In: *European Symposium on Programming*. Springer. 1994, pp. 302–316. DOI: 10.1007/3-540-57880-3_20.

[32] Shin-ya Katsumata. "Parametric effect monads and semantics of effect systems." In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 633–646. DOI: 10.1145/2535838.2535846.

[33] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. "Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl)." In: *SIGPLAN Not.* 40.9 (Sept. 2005), 192–203. ISSN: 0362-1340. DOI: 10.1145/1090189.1086390.

[34] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. "Resource-Guided Program Synthesis." In: *CoRR* abs/1904.07415 (2019). arXiv: 1904.07415. URL: http://arxiv.org/abs/1904.07415.

[35] Chuck Liang and Dale Miller. "Focusing and polarization in linear, intuitionistic, and classical logics." In: *Theoretical Computer Science* 410.46 (2009), pp. 4747–4768.

[36] *Logic Programming with Linear Logic*. Accessed 19th June 2020. URL: http://www.cs.rmit.edu.au/lygon/..

[37] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. "A Generic Deriving Mechanism for Haskell." In: *SIGPLAN Not.* 45.11 (Sept. 2010), 37–48. ISSN: 0362-1340. DOI: 10.1145/2088456.1863529. URL: https://doi.org/10.1145/2088456.1863529.

[38] Zohar Manna and Richard Waldinger. "A deductive approach to program synthesis." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1 (1980), pp. 90–121.

[39] Conor McBride. "I Got Plenty o' Nuttin'." In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella. Cham: Springer International Publishing, 2016, pp. 207–233. ISBN: 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_12.

[40] Benjamin Moon, Harley Eades III, and Dominic Orchard. "Graded Modal Dependent Type Theory." In: *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European*

Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings. Ed. by Nobuko Yoshida. Vol. 12648. Lecture Notes in Computer Science. Springer, 2021, pp. 462–490. DOI: 10.1007/978-3-030-72019-3_17. URL: https://doi.org/10.1007/978-3-030-72019-3_17.

[41] Leonardo de Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver." In: vol. 4963. Apr. 2008, pp. 337–340.

[42] Dominic A. Orchard, Tomas Petricek, and Alan Mycroft. "The semantic marriage of monads and effects." In: *CoRR* abs/1401.5391 (2014). arXiv: 1401.5391. URL: http://arxiv.org/abs/1401.5391.

[43] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. "Quantitative program reasoning with graded modal types." In: *PACMPL* 3.ICFP (2019), 110:1–110:30. DOI: 10.1145/3341714.

[44] Peter-Michael Osera and Steve Zdancewic. "Type-and-Example-Directed Program Synthesis." In: *SIGPLAN Not.* 50.6 (June 2015), 619–630. ISSN: 0362-1340. DOI: 10.1145/2813885.2738007.

[45] Tomas Petricek, Dominic Orchard, and Alan Mycroft. "Coeffects: a calculus of context-dependent computation." In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. ACM. 2014, pp. 123–135. DOI: 10.1145/2692915.2628160.

[46] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. "Program Synthesis from Polymorphic Refinement Types." In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, 522–538. ISBN: 9781450342612. DOI: 10.1145/2908080.2908093. URL: https://doi.org/10.1145/2908080.2908093.

[47] John Power and Hiroshi Watanabe. "Combining a monad and a comonad." In: *Theoretical Computer Science* 280.1-2 (2002), pp. 137–162. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(01)00024-X.

[48] Calvin Smith and Aws Albarghouthi. "Synthesizing Differentially Private Programs." In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341698.

[49] Ross Street. “The formal theory of monads.” In: *Journal of Pure and Applied Algebra* 2.2 (1972), pp. 149–168. DOI: 10.1016/0022-4049(72)90019-9.

[50] Tarmo Uustalu and Varmo Vene. “The Essence of Dataflow Programming.” In: *Lecture Notes in Computer Science* 4164 (November 2006), pp. 135–167. DOI: 10.1007/11894100_5.

[51] Uma Zalakain and Ornela Dardha. “Pi with leftovers: a mechanisation in Agda.” In: *arXiv preprint arXiv:2005.05902* (2020).

Part II

APPENDIX

A

PROOFS

A.1 PROOFS FOR THE ADDITIVE AND SUBTRACTIVE LINEAR-BASE CALCULI

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 A.1.1 (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 A.1.1 $(\Gamma + (\Gamma' - \Gamma'') \sqsubseteq (\Gamma + \Gamma') - \Gamma'')$**.**

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 2.3.1 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 2.3.1 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 2.3.1 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 2.3.1 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:

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

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 2.3.1 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 A.1.2 ($(\Gamma - \Gamma') + \Gamma' \sqsubseteq \Gamma$)**.**

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

1. $\Gamma' = \varnothing$
   We have:

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

   From definition 3.3.1, we know that:

   $$\Gamma - \varnothing = \Gamma$$

   and from definition 2.3.1, we know:

   $$\Gamma + \varnothing = \Gamma$$

   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 2.3.1, 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 A.1.3 (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 A.1.4. *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 **??**, 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.

□

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* **??** *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:

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

2. Case $\textsc{GrVar}^-$
   Matching the form of the lemma, we have the derivation:

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

   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:

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

   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:

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

4. Case $\multimap_L^-$
Matching the form of the lemma, the application derivation is:

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

By induction, we have that:

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

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

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

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

We can thus construct the following typing derivation, making use of of the admissibility of linear substitution (Lemma 2.4.1):

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

From Lemma **??**, we have that

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

and from Lemma **??**, 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 $\textsc{Approx}$ 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 $\square_R^-$
The synthesis rule for boxing can be constructed as:

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

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:

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

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

$$\frac{\dfrac{[\Gamma - \Delta] \vdash t : A}{r \cdot [\Gamma - \Delta] \vdash [t] : \square_r A}\ \textsc{Pr} \quad \text{Lem. } \mathbf{??}}{\Gamma - (\Gamma - (r \cdot (\Gamma - \Delta))) \vdash [t] : \square_r A}\ \textsc{Approx}$$

Satisfying the goal of the lemma.

6. Case $\square_L^-$
The synthesis rule for unboxing has the form:

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

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:

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

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:

$$\Gamma - \Delta_1 \vdash t_1 : A \qquad \text{(ih1)}$$
$$\Delta_1 - \Delta_2 \vdash t_2 : B \qquad \text{(ih2)}$$

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 **??**, we have that:

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

and from Lemma **??**, 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:

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

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:

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

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:

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

   By induction we have:

   $$\Gamma - \Delta \vdash t : A \qquad \text{(ih1)}$$

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

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

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

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

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

    By induction:

    $$(\Gamma, x_2 : A) - \Delta_1 \vdash t_1 : C \qquad \text{(ih)}$$
    $$(\Gamma, x_3 : B) - \Delta_2 \vdash t_2 : C \qquad \text{(ih)}$$

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^-

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

By Lemma **??** we have that $\emptyset \sqsubseteq \Gamma - \Gamma$ then we have:

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

Matching the goal

12. Case 1_L^-

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

By induction we have:

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

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 $\textsc{der}^-$we have $r \sqsupseteq (s + 1)$.

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

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:

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

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

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

2. Case GrVar$^+$

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

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

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

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

3. Case $\multimap_R^+$

We thus have the derivation:

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

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:

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

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:

$$\begin{array}{r} \Delta_1, x_2 : B \vdash t_1 : C \\ \Delta_2 \vdash t_2 : A \end{array}$$

We can thus construct the following typing derivation, making use of the admissibility of linear substitution (Lemma 2.4.1):

$$\dfrac{\begin{array}{c} \dfrac{\dfrac{}{x_1 : A \multimap B \vdash x_1 : A \multimap B}\,\textsc{var} \qquad \Delta_2 \vdash t_2 : A}{\Delta_2, x_1 : A \multimap B \vdash x_1\, t_2 : B}\,\textsc{app} \\ \Delta_1, x_2 : B \vdash t_1 : C \end{array}}{(\Delta_1 + \Delta_2), x_1 : A \multimap B \vdash [(x_1\, t_2)/x_2]t_1 : C}\,\text{(L. 2.4.1)}$$

5. Case $\Box_R^+$
   The synthesis rule for boxing can be constructed as:

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

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:

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

6. Case $\textsc{der}^+$
   From the dereliction rule we have:

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

By induction we get:

$$\Delta, y : A \vdash t : B \tag{ih}$$

Case on $x \in \Delta$

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

  Then by admissibility of contraction we can derive:

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

  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:

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

   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:

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

   By induction on the premises we have that:

$$\Delta_1 \vdash t_1 : A \qquad \text{(ih1)}$$
$$\Delta_2 \vdash t_2 : B \qquad \text{(ih2)}$$

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

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

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:

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

By induction on the premises we have that:

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

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

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

Likewise, for the $\oplus 2_R^+$we have the synthesis rule:

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

By induction on the premises we have that:

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

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

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

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

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

By induction on the premises we have that:

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

$$\Delta_2, x_3 : B \vdash t_2 : C \qquad \text{(ih2)}$$

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

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

12. Case 1_R^+
    The synthesis rule for unit introduction has the form:

$$\dfrac{}{\Gamma \vdash 1 \Rightarrow^+ ();\, \varnothing}\;1_R^+$$

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

$$\dfrac{}{\varnothing \vdash () : 1}\;1$$

13. Case 1_L^+
    The synthesis rule for unit elimination has the form:

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

By induction on the premises we have that:

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

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

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

□

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:

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

   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 2.4.1):

$$\dfrac{\dfrac{\dfrac{}{x_1 : A \multimap B \vdash x_1 : A \multimap B}\ \textsc{var} \qquad \Delta_2 \vdash t_2 : A}{\Delta_2, x_1 : A \multimap B \vdash x_1\, t_2 : B}\ \textsc{app} \qquad \Delta_1, x_2 : B \vdash t_1 : C}{(\Delta_1 + \Delta_2), x_1 : A \multimap B \vdash [(x_1\, t_2)/x_2]t_1 : C}\ (\text{L. } 2.4.1)$$

2. Case $\otimes_R'^{+}$

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

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

   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:

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

□

Lemma A.1.5 (Soundness of focusing for subtractive synthesis).
For all contexts Γ, Ω *and types* A *then:*

$$\begin{array}{llll}
1.\ \textit{Right Async}: & \Gamma;\Omega \vdash A \Uparrow \Rightarrow^- t \mid \Delta & \Longrightarrow & \Gamma,\Omega \vdash A \Rightarrow^- t \mid \Delta \\
2.\ \textit{Left Async}: & \Gamma;\Omega \Uparrow \vdash C \Rightarrow^- t \mid \Delta & \Longrightarrow & \Gamma,\Omega \vdash C \Rightarrow^- t \mid \Delta \\
3.\ \textit{Right Sync}: & \Gamma;\varnothing \vdash A \Downarrow \Rightarrow^- t \mid \Delta & \Longrightarrow & \Gamma \vdash A \Rightarrow^- t \mid \Delta \\
4.\ \textit{Left Sync}: & \Gamma;x : A \Downarrow \vdash C \Rightarrow^- t \mid \Delta & \Longrightarrow & \Gamma,x : A \vdash C \Rightarrow^- t \mid \Delta \\
5.\ \textit{Focus Right}: & \Gamma;\Omega \Uparrow \vdash C \Rightarrow^- t \mid \Delta & \Longrightarrow & \Gamma \vdash C \Rightarrow^- t \mid \Delta \\
6.\ \textit{Focus Left}: & \Gamma,x : A;\Omega \Uparrow \vdash C \Rightarrow^- t \mid \Delta & \Longrightarrow & \Gamma \vdash C \Rightarrow^- t \mid \Delta
\end{array}$$

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:

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

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:

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

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:

$$\frac{\Gamma;\Omega \Uparrow \vdash C \Rightarrow^- t \mid \Delta \qquad C \text{ not right async}}{\Gamma;\Omega \vdash C \Uparrow \Rightarrow^- t \mid \Delta} \Uparrow_R^-$$

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:

$$\frac{\begin{array}{c}\Gamma; \Omega, x_1 : A, x_2 : B \Uparrow \vdash C \Rightarrow^- t_2 \mid \Delta \\ x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|\end{array}}{\Gamma; \Omega, x_3 : A \otimes B \Uparrow \vdash C \Rightarrow^- \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta}\ \otimes_L^-$$

   By induction on the first premise, we have that:

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

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

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

   b) Case $\oplus_L^-$
   In the case of the left asynchronous rule for sum elimination, the synthesis rule has the form:

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

   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:

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

c) Case 1_L^-
In the case of the left asynchronous rule for unit elimination, the synthesis rule has the form:

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

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:

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

d) Case $\Box_L^-$
In the case of the left asynchronous rule for graded modality elimination, the synthesis rule has the form:

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

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:

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

e) Case DER$^-$
In the case of the left asynchronous rule for dereliction, the synthesis rule has the form:

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

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:

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

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:

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

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:

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

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

$$\Gamma \vdash A \Rightarrow^- t_1 \mid \Delta_1 \tag{ih1}$$

$$\Delta_1 \vdash B \Rightarrow^- t_2 \mid \Delta_2 \tag{ih2}$$

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:

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

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

By induction on the premises of these rules, we have that:

$$\Gamma \vdash A \Rightarrow^- t \mid \Delta \qquad \text{(ih1)}$$

$$\Gamma \vdash B \Rightarrow^- t \mid \Delta \qquad \text{(ih2)}$$

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:

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

From which, we can construct the following instatiation of the 1^-_R synthesis rule in the non-focusing calculus:

$$\frac{}{\Gamma, \Omega \vdash 1 \Rightarrow^- () \mid \Gamma} 1^-_R$$

d) Case $\Box^-_R$
In the case of the right synchronous rule for graded modality introduction, the synthesis rule has the form:

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

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:

$$\frac{\Gamma; \varnothing \vdash A \Uparrow \Rightarrow^{-} t \mid \Delta}{\Gamma; \varnothing \vdash A \Downarrow \Rightarrow^{-} t \mid \Delta}\ \Downarrow_R^{-}$$

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:

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

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:

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

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 GrVar$^-$
In the case of the left synchronous rule for graded variable synthesis, the synthesis rule has the form:

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

From which, we can construct the following instatiation of the 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:

$$\frac{\Gamma; x : A \Uparrow \vdash C \Rightarrow^- t \mid \Delta \qquad \text{A not atomic and not left sync}}{\Gamma; x : A \Downarrow \vdash C \Rightarrow^- t \mid \Delta}\ \Downarrow_L^-$$

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:

$$\frac{\Gamma;\varnothing \vdash C \Downarrow \Rightarrow^{-} t \mid \Delta \qquad \text{C not atomic}}{\Gamma;\varnothing \Uparrow \vdash C \Rightarrow^{-} t \mid \Delta}\ \textsc{focus}_R^{-}$$

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:

$$\frac{\Gamma; x : A \Downarrow \vdash C \Rightarrow^{-} t \mid \Delta}{\Gamma, x : A;\varnothing \Uparrow \vdash C \Rightarrow^{-} t \mid \Delta}\ \textsc{focus}_L^{-}$$

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.

□

Lemma A.1.6 (Soundness of focusing for additive synthesis). *For all contexts* Γ, Ω *and types A then:*

$$\begin{array}{lllll}
1. & \textit{Right Async}: & \Gamma;\Omega \vdash A \Uparrow \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma, ,\Omega \vdash A \Rightarrow^{+} t;\ \Delta \\
2. & \textit{Left Async}: & \Gamma;\Omega \Uparrow \vdash C \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma, ,\Omega \vdash C \Rightarrow^{+} t;\ \Delta \\
3. & \textit{Right Sync}: & \Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma \vdash A \Rightarrow^{+} t;\ \Delta \\
4. & \textit{Left Sync}: & \Gamma; x : A \Downarrow \vdash C \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma, x : A \vdash C \Rightarrow^{+} t;\ \Delta \\
5. & \textit{Focus Right}: & \Gamma;\Omega \Uparrow \vdash C \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma \vdash C \Rightarrow^{+} t;\ \Delta \\
6. & \textit{Focus Left}: & \Gamma, x : A;\Omega \Uparrow \vdash C \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma \vdash C \Rightarrow^{+} t;\ \Delta
\end{array}$$

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:

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

By induction on the premise, we have that:

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

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:

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

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:

$$\frac{\Gamma; \Omega \Uparrow \vdash C \Rightarrow t \mid \Delta \qquad C \text{ not right async}}{\Gamma; \Omega \vdash C \Uparrow \Rightarrow t \mid \Delta}\ \Uparrow^{+}_{R}$$

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:

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

By induction on the premise, we have that:

$$(\Gamma, \Omega), x_1 : A, x_2 : B \vdash C \Rightarrow^{+} t_2;\ \Delta, x_1 : A, x_2 : B \quad \text{(ih)}$$

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:

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

b) Case $\oplus_L^+$
In the case of the left asynchronous rule for sum elimination, the synthesis rule has the form:

$$\dfrac{\begin{array}{c}\Gamma;\Omega, x_2 : A \Uparrow \vdash C \Rightarrow t_1 \mid \Delta_1, x_2 : A \\ \Gamma;\Omega, x_3 : B \Uparrow \vdash C \Rightarrow t_2 \mid \Delta_2, x_3 : B\end{array}}{\Gamma;\Omega, x_1 : A \oplus B \Uparrow \vdash C \Rightarrow^- \textbf{case}\ x_1\ \textbf{of}\ \textbf{inl}\ x_2 \rightarrow t_1;\ \textbf{inr}\ x_3 \rightarrow t_2 \mid \Delta_1 \sqcup \Delta_2, x_1 : A \oplus}$$

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:

$$\dfrac{\begin{array}{c}(\Gamma, \Omega), x_2 : A \vdash C \Rightarrow^+ t_1;\ \Delta_1, x_2 : A \\ (\Gamma, \Omega), x_3 : B \vdash C \Rightarrow^+ t_2;\ \Delta_2, x_3 : B\end{array}}{\Gamma, (\Omega, x_1 : A \oplus B) \vdash C \Rightarrow^+ \textbf{case}\ x_1\ \textbf{of}\ \textbf{inl}\ x_2 \rightarrow t_1;\ \textbf{inr}\ x_3 \rightarrow t_2 \mid (\Delta_1 \sqcup \Delta_2), x_1 : A \oplus}$$

c) Case 1_L^+
In the case of the left asynchronous rule for unit elimination, the synthesis rule has the form:

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

By induction on the premise, we have that:

$$\Gamma \vdash C \Rightarrow^+ t;\ \Delta \qquad \text{(ih)}$$

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:

$$\dfrac{\Gamma \vdash C \Rightarrow^+ t;\ \Delta}{\Gamma, x : 1 \vdash C \Rightarrow^+ \textbf{let}\,() = x\,\textbf{in}\,t;\ \Delta, x : 1}\ \text{L}1^+$$

d) Case $\Box_L^+$
In the case of the left asynchronous rule for graded modality elimination, the synthesis rule has the form:

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

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:

$$\dfrac{\begin{array}{c}(\Gamma, \Omega), x_2 :_r A \vdash B \Rightarrow^+ t;\ \Delta \\ \textit{if } x_2 :_s A \in \Delta \textit{ then } s \sqsubseteq r \textit{ else } 0 \sqsubseteq r\end{array}}{\Gamma, (\Omega, x_1 : \Box_r A) \vdash B \Rightarrow^+ \textbf{let}\,[x_2] = x_1\,\textbf{in}\,t;\ (\Delta \backslash x_2), x_1 : \Box_r A}\ \mathrm{L}\Box^+$$

e) Case DER$^+$
In the case of the left asynchronous rule for dereliction, the synthesis rule has the form:

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

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:

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

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:

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

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:

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

   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:

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

   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:

$$\frac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta}{\Gamma;\varnothing \vdash A \oplus B \Downarrow \Rightarrow \mathbf{inl}\, t \mid \Delta}\ \oplus 1_L^+$$

$$\frac{\Gamma;\varnothing \vdash B \Downarrow \Rightarrow t \mid \Delta}{\Gamma;\varnothing \vdash A \oplus B \Downarrow \Rightarrow \mathbf{inr}\, t \mid \Delta}\ \oplus 2_L^+$$

   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:

$$\frac{\Gamma \vdash A \Rightarrow^+ t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inl}\, t;\ \Delta}\ \mathrm{R}\oplus_1^+$$

$$\frac{\Gamma \vdash B \Rightarrow^+ t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inr}\, t;\ \Delta}\ \mathrm{R}\oplus_2^+$$

c) Case 1^+_R
In the case of the right synchronous rule for unit introduction, the synthesis rule has the form:

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

From which, we can construct the following instantiation of the 1^+_R synthesis rule in the non-focusing calculus:

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

d) Case $\Box^+_R$
In the case of the right synchronous rule for graded modality introduction, the synthesis rule has the form:

$$\frac{\Gamma;\emptyset \vdash A \Uparrow \Rightarrow t \mid \Delta}{\Gamma;\emptyset \vdash \Box_r A \Downarrow \Rightarrow [t] \mid r \cdot \Delta}\ \Box^+_R$$

By induction on the premise, we have that:

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

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:

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

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:

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

By induction on the premise, we have that:

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

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:

$$\dfrac{\begin{array}{c}\Gamma; x_2 : B \Downarrow \vdash C \Rightarrow t_1 \mid \Delta_1, x_2 : B \\ \Gamma; \varnothing \vdash A \Downarrow \Rightarrow t_2 \mid \Delta_2\end{array}}{\Gamma; x_1 : A \multimap B \Downarrow \vdash C \Rightarrow [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B} \multimap^{+}_{L}$$

   By induction on the first premise, we have that:

$$\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1; \Delta_1, x_2 : B \qquad \text{(ih1)}$$

   from case 4 of the lemma. By induction on the second premise, we have that:

$$\Gamma \vdash A \Rightarrow^{+} t_2; \Delta_2 \qquad \text{(ih2)}$$

   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:

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

   b) Case $\textsc{LinVar}^{+}$
   In the case of the left synchronous rule for linear variable synthesis, the synthesis rule has the form:

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

   From which, we can construct the following instantiation of the $\textsc{LinVar}^{+}$ in the non-focusing calculus:

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

   c) Case $\textsc{GrVar}^{+}$
   In the case of the left synchronous rule for graded variable synthesis, the synthesis rule has the form:

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

From which, we can construct the following instantiation of the 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:

$$\frac{\begin{array}{c}\Gamma; x : A \Uparrow \vdash C \Rightarrow t \mid \Delta \\ \text{A not atomic and not left sync}\end{array}}{\Gamma; x : A \Downarrow \vdash C \Rightarrow t \mid \Delta}\ \Downarrow_L^+$$

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:

$$\frac{\Gamma; \emptyset \vdash C \Downarrow \Rightarrow t \mid \Delta \qquad \text{C not atomic}}{\Gamma; \emptyset \Uparrow \vdash C \Rightarrow t \mid \Delta}\ \textsc{focus}_R^+$$

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:

$$\frac{\Gamma; x : A \Downarrow \vdash C \Rightarrow t \mid \Delta}{\Gamma, x : A; \emptyset \Uparrow \vdash C \Rightarrow t \mid \Delta}\ \textsc{focus}_L^+$$

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 A.1.7 (Soundness of focusing for additive pruning synthesis)**.** *For all contexts* Γ*,* Ω *and types* A *then:*

$$\begin{array}{llcl}
1.\ \mathit{Right\ Async}: & \Gamma;\Omega \vdash A \Uparrow \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma,,\Omega \vdash A \Rightarrow^{+} t;\, \Delta \\
2.\ \mathit{Left\ Async}: & \Gamma;\Omega \Uparrow \vdash C \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma,,\Omega \vdash C \Rightarrow^{+} t;\, \Delta \\
3.\ \mathit{Right\ Sync}: & \Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma \vdash A \Rightarrow^{+} t;\, \Delta \\
4.\ \mathit{Left\ Sync}: & \Gamma;x : A \Downarrow \vdash C \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma, x : A \vdash C \Rightarrow^{+} t;\, \Delta \\
5.\ \mathit{Focus\ Right}: & \Gamma;\Omega \Uparrow \vdash C \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma \vdash C \Rightarrow^{+} t;\, \Delta \\
6.\ \mathit{Focus\ Left}: & \Gamma, x : A;\Omega \Uparrow \vdash C \Rightarrow t \mid \Delta & \Longrightarrow & \Gamma \vdash C \Rightarrow^{+} t;\, \Delta
\end{array}$$

Proof.

1. Case: 1. Right Async: The proofs for right asynchronous rules are equivalent to those of lemma (**??**)

2. Case 2. Left Async: The proofs for left asynchronous rules are equivalent to those of lemma (**??**)

3. Case 3. Right Sync: The proofs for right synchronous rules are equivalent to those of lemma (**??**), 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:

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

   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 (**??**), 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:

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

By induction on the first premise, we have that:

$$\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1;\ \Delta_1, x_2 : B \qquad \text{(ih1)}$$

from case 4 of the lemma. By induction on the second premise, we have that:

$$\Gamma \vdash A \Rightarrow^{+} t_2;\ \Delta_2 \qquad \text{(ih2)}$$

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:

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

5. Case 5. Right Focus: focus_R^{+} - The proof for right focusing rule is equivalent to that of lemma (**??**)

6. Case 6. Left Focus: focus_L^{+} - The proof for left focusing rule is equivalent to that of lemma (**??**)

□

DECLARATION

Put your declaration here.

Canterbury, November 2023

Jack Oliver Hughes

colophon

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`

Thank you very much for your feedback and contribution.

Final Version as of October 31, 2023 (classicthesis v4.6).