Program Synthesis from Graded Types

Jack Hughes[1] (✉) and Dominic Orchard[1,2]

[1] University of Kent, Canterbury, UK
joh6@kent.ac.uk
[2] University of Cambridge, Cambridge, UK

Abstract. Graded type systems are a class of type system for fine-grained quantitative reasoning about data-flow in programs. Through the use of resource annotations (or *grades*), a programmer can express various program properties at the type level, reducing the number of typeable programs. These additional constraints on types lend themselves naturally to type-directed *program synthesis*, where this information can be exploited to constrain the search space of programs. We present a synthesis algorithm for a graded type system, where grades form an arbitrary pre-ordered semiring. Harnessing this grade information in synthesis is non-trivial, and we explore some of the issues involved in designing and implementing a resource-aware program synthesis tool. In our evaluation we show that by harnessing grades in synthesis, the majority of our benchmark programs (many of which involve recursive functions over recursive ADTs) require less exploration of the synthesis search space than a purely type-driven approach and with fewer needed input-output examples. This *type-and-graded-directed* approach is demonstrated for the research language Granule but we also adapt it for synthesising Haskell programs that use GHC's linear types extension.

1 Introduction

Type-directed program synthesis is a technique for synthesising programs from user-provided type specifications. The technique has a long history intertwined with proof search, thanks to the Curry-Howard correspondence [37, 22]. We present a program synthesis approach that leverages the information of *graded type systems* that track and enforce program properties related to data flow. Our approach follows the concept 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 express this in terms of a synthesis *judgement* akin to 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. A calculus of synthesis *rules* for inductively defines the above synthesis judgement for each type former of a language. For example, we may define a synthesis rule for standard product types in the following way:

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

S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 83–112, 2024.
https://doi.org/10.1007/978-3-031-57262-3_4

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.

Depending on the context, there may be many possible combinations of assumptions to synthesise a pair. Consider the following type and partial program with a *hole* (marked ?) specifying a position to perform program synthesis:

$$f : A \to A \to A \to A \times A \qquad f\ x\ y\ z = ?$$

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 $\times_{\text{INTRO}}$ rule yields:

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

Even in this simple setting, the number of possibilities starts to become unwieldy: there are 3^2 possible candidate programs based on combinations of x, y and z. We thus wish to constrain the number of choices required by the synthesis algorithm. Many systems achieve this by allowing the user to specify additional information about the desired program behaviour. For example, recent work extends type-directed synthesis to refinement types [50], cost specifications [35], differential privacy [52], ownership information [16], example-guided synthesis [15, 2] and examples integrated with types [17, 47]. The general idea is that the proof search / program synthesis procedure can be pruned and refined given more information, whether richer types, additional examples, or behavioural specifications.

We instead leverage the information contained in *graded type systems* which constrain how data can be used by a program and thus reduce the number of possible synthesis choices. Our hypothesis is that grade-and-type-directed synthesis reduces the number of paths that need to be explored and the number of input-output examples that are needed, thus potentially speeding up synthesis.

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, disallowing arbitrary copying and discarding [20]. Non-linearity is captured by 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 [21], e.g., $!_{\leq 4}A$ represents a value A which may be used up to 4 times. Various works have generalised BLL, resulting in *graded* type systems in which these indices are drawn from an arbitrary pre-ordered semiring [12, 19, 49, 1, 14, 5, 39]. This allows numerous program properties to be tracked and enforced statically. Such systems are being deployed in language implementations, forming the basis of Haskell's linear types extension [8], Idris 2 [11], and the language Granule [45].

Returning to our example in a graded setting, the function's parameters now have *grades* that we choose, for the sake of example, to be particular natural

numbers describing the exact number of times the parameters must be used:

$$f : A^2 \to A^0 \to A^0 \to A \times A \qquad f\ x\ y\ z = ?$$

The first A is annotated with a grade 2 meaning it *must* be used twice. The types of y and z are graded with 0, enforcing zero usage, i.e., they cannot be used in the body of f. The result is that there is only one (normal form) inhabitant for this type: (x, x). For synthesis, the other assumptions will not even be considered, allowing pruning of branches which use resources in a way which violates the grades. Natural number grades in this example explain how many times a value can be used, but we may instead represent different program properties such as sensitivity, strictness, or security levels for tracking non-interference, all of which are well-known instances of graded type systems [45, 18, 1]. These examples are all graded presentations of *coeffects*, tracking how a programs uses its context, in contrast with graded types for *effects* [46, 32] which are not considered here.

In prior work, we built on proof search for linear logic [25], developing a program synthesis technique for a linear type theory with graded modalities $\Box_r A$ (where r is drawn from a semiring) and non-recursive types [27], which we refer to as LGM i.e., *linear-graded-modal*. We adapt some of these ideas to a setting which does not have a linear basis, but rather a type system where grades are pervasive (such as the core of Haskell's linear types extension [8]) alongside recursive algebraic data types and input-output example specifications.

We make the following contributions:

- We define a synthesis calculus for a core graded type system, adapting the context management scheme of LGM to a fully graded setting (rather than the linear setting) and also addressing recursion, recursive types, and user-defined ADTs, none of which were considered in previous work. Synthesised is proved sound, i.e., synthesised programs are typed by the goal type.
- We implemented both the core type system as an extension of Granule and implemented the synthesis calculus algorithmically.[1] We elide full details of the implementation but explain its connection to the formal development.
- We extend the Granule language to include input-output examples as specifications with first-class syntax (that is type checked), which complements the synthesis algorithm and helps guide synthesis. This also aids our evaluation.
- 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 [47].
- Leveraging our calculus and implementation, we provide a prototype tool for synthesising Haskell programs that use GHC 9's linear types extension.

Roadmap Section 2 gives a brief overview of proof search in resourceful settings, recalling the 'resource management problem'. Section 3 then defines a core calculus as the target of our synthesis approach. This type system closely resembles various other graded systems [49, 39, 5, 1] including the core of Linear Haskell [8]. We implemented this system as a language extension of Granule [45].

[1] Available at: `github.com/granule-project/granule/releases/tag/v0.9.3.0`

Section 4 presents a calculus of synthesis rules for our language, showing how grades enforce resource usage potentially leading to pruning of the search space of candidate programs. We also discuss some details of the implementation of our tool. We observe the close connection between synthesis in a graded setting and automated theorem proving for linear logic, allowing us to exploit existing optimisation techniques, such as the idea of a focused proof [4].

Section 5 evaluates our implementation on a set of 46 benchmarks, including several non-trivial programs which use algebraic data types and recursion.

Section 6 demonstrates the practicality and versatility of our approach by retargeting our algorithm to synthesise programs in Haskell from type signatures that use GHC's *linear types* extension (which is a graded type system [8]).

2 Overview of Resourceful Program Synthesis

Section 1 discussed synthesising pairs and how graded types could control the number of times assumptions are used in a synthesised term. In a linear or graded setting, synthesis must handle the *resource management problem* [24, 13]: how do we give a resourceful accounting to the context during synthesis, respecting its constraints? We overview the main ideas for addressing this problem.

Section 1 considered (Cartesian) product types $\times$, but we now switch to the *multiplicative product* of linear types, which has the typing rule [20]:

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

Each subterm is typed by different contexts, which are combined by *disjoint* union: a pair cannot be formed if variables are shared between Γ_1 and Γ_2, preventing the structural behaviour of *contraction* where variables appear in multiple subterms. Naïvely converting 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, this 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 a context Γ and a goal $A \otimes B$, we have to split Γ into disjoint subparts Γ_1 and Γ_2 such that $\Gamma = \Gamma_1, \Gamma_2$ in order to pass 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 [25], refined by Cervesato et al. [13], where proof search has an *input context* of available resources and an *output context* of the remaining resources, which we write as judgments $\Gamma \vdash A \Rightarrow^- t \mid \Gamma'$ for input context Γ and output context Γ'. Synthesis for multiplicative products then becomes:

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

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

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

This avoids the problem of splitting the input context, facilitating efficient proof search for linear types. LGM adapted this idea to linear types augmented with graded modalities [27]. We call the above approach *subtractive resource management* due to its similarity to *left-over* type-checking for linear types [3, 54]. In a graded modal setting however this approach is costly [27].

Graded type systems, as considered here, have typing contexts in which free variables are assigned a type and a *grade*: an element of a semiring. For example, the semiring of natural numbers describes how many times an assumption can be used, in contrast to linear assumptions which must be used exactly once, e.g., the context $x :_2 A, y :_0 B$ says that x must be used twice but y cannot be used. The literature contains many example semirings for tracking other properties as graded types, e.g., security labels [18, 1], intervals of usage [45], and hardware schedules [19]. In a graded setting, the subtractive approach is problematic though as there is not necessarily a notion of subtraction for grades.

Consider the above example but for a context with grades r and s on the variables. Using a variable to synthesise a subterm no longer results in that variable being 'left out' of the output context. Instead a new grade is given in the output context relating to the input with a constraint capturing the usage:

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

In the first premise, x has grade r in the input context and x is synthesised for the goal, thus the output context has some grade r' where $r' + 1 = r$, denoting a use of x by the 1 element of the semiring. The second premise is similar.

For the natural numbers, if $r = s = 1$ then the above constraints are satisfied by $r' = s' = 0$. In general, subtractive synthesis for graded types requires solving many such existential equations over semirings, which introduces a new source of non-determinism as there can be more than one solution. LGM implemented this approach, leveraging SMT solving in the context of the Granule language, but show that a dual *additive* approach has better performance. In the additive approach, output contexts describe what was *used* instead of what is *left*. To synthesise a term with multiple subterms (e.g. pairs), the output contexts of each premise are added using the semiring addition applied pointwise on contexts to produce the conclusion output. For pairs this looks like:

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

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

$$\frac{x :_r A, y :_s B \vdash A \Rightarrow^+ x \mid x :_1 A, y :_0 B \quad x :_r A, y :_s B \vdash A \Rightarrow^+ x \mid x :_1 A, y :_0 B}{x :_r A, y :_s B \vdash A \otimes A \Rightarrow^+ (x, x) \mid x :_{1+1} A, y :_0 B}\ \otimes^+_{\text{INTRO}} \qquad \text{(example)}$$

Synthesis rules for binders check whether the output context describes use that is within the grades given by Γ, i.e., that synthesised terms are *well-resourced*.

Both subtractive and additive approaches avoid having to split the incoming context Γ prior to synthesising subterms. In LGM, we evaluated both resource management strategies in a synthesis tool for a subset of Granule's 'linear base' system, finding that in most cases, the additive strategy was more efficient for use in program synthesis with grades as it involves solving less complex predicates; the subtractive approach typically incurs higher overhead due to the existentially-derived notion of subtraction seen above. We therefore take the additive approach to resource management.

LGM developed our approach for the linear λ-calculus with products, coproducts, and semiring-graded modalities. Here, we instead consider a graded calculus without a linear base but where all assumptions are graded and function types therefore incorporate a grade. Furthermore, our approach permits synthesis for user-defined recursive ADTs to address more real-world problems.

3 Core Calculus

We define a core language with graded types, drawing from the coeffect calculus of Petricek et al. [49], Quantitative Type Theory (QTT) [39, 5] and other graded dependent type theories [42] (omitting dependent types from our language), the calculus of Abel and Bernardy [1], and the core of the linear types extension to Haskell [8]. This calculus shares much in common with languages based on linear types, such as the graded monadic-comonadic calculus of [18], generalisations of Bounded Linear Logic [12, 19], and Granule [45] in its original 'linear base' form.

Our target calculus extends the λ-calculus with grades and a graded necessity modality as well as recursive algebraic data types. Parameterising the calculus is a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ where pre-ordering requires that $+$ and $*$ are monotonic wrt. $\sqsubseteq$. Throughout r, s range over $\mathcal{R}$. The syntax of types is:

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

The function space $A^r \to B$ annotates the input type with a *grade* $r \in \mathcal{R}$. The graded necessity modality $\Box_r A$ is similarly annotated/indexed with a grade r. Type constructors K include the multiplicative linear products and units, additive coproducts, and is extended by names of user-defined ADTs in the implementation. Constructors are applied to zero or more type parameters written $\overline{A}$. Recursive types $\mu X.A$ are equi-recursive with type recursion variables X. Data

constructors and other top-level definitions are typed by type schemes τ (rank-1 polymorphic types), which bind a set of kind-annotated universally quantified type variables $\overline{\alpha : \kappa}$ à la ML [40]. Subsequently, types may contain type variables α. Kinds κ are standard, given in the appendix [28].

The term syntax comprises the λ-calculus, a *promotion* construct $[t]$ which introduces a graded modality, data constructors $(C\, t_1 \ldots t_n)$, and elimination by **case** expressions with patterns p, where $[p]$ eliminates graded modalities:

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

Example 1. In the type system (below), the k-combinator is typed as on the left:

$$k : A^1 \to B^0 \to A \qquad k' : (A \times \Box_0 B)^r \to \Box_r A$$
$$k = \lambda x.\lambda y.x \qquad k' = \lambda p.\mathbf{case}\ p\ \mathbf{of}\ (x, y) \mapsto \mathbf{case}\ y\ \mathbf{of}\ [y'] \mapsto [x]$$

On the right, an uncurried version uses graded modalities. The argument pair uses a graded modality to capture that the B part is not used. This graded modal argument is eliminated by the second **case** with pattern $[y']$ binding y' with grade 0, indicating it is unused. The return result is of graded modal type with some grade r which is introduced by promotion $[x]$. Promotion propagates its grade to its dependencies, i.e., the parameter p must also have grade r.

A useful semiring is of *security levels* [18, 1], e.g., $\mathcal{R} = \{\mathsf{Private}, \mathsf{Public}\}$ where $\mathsf{Private} \sqsubseteq \mathsf{Public}$, $+ = \wedge$ with $0 = \mathsf{Private}$, and $* = \vee$ with $1 = \mathsf{Public}$. In the above example, the second argument to k would thus be $\mathsf{Private}$. If the return result of k' is for public consumption, i.e., $r = \mathsf{Public}$, then the argument must also be public, with the private component B not usable in the result.

Figure 1 defines the typing judgments of the form $\Sigma; \Gamma \vdash t : A$ assigning a type A to a term t under type variables Σ. For such judgments we say that t is both *well typed* and *well resourced* to highlight the role of grading in accounting for resource use via the grades. Contexts Γ are given by:

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

That is, a context may be empty $\emptyset$ or extended with a *graded* assumption $x :_r A$. Graded assumptions must be used in a way which adheres to the grade r. Structural exchange is permitted, allowing a context to be arbitrarily reordered. A global context D parameterises the system, containing top-level definitions and data constructors annotated with type schemes. A context of kind annotated type variables Σ is used for kinding and when instantiating a type scheme from D. Appendix A gives the (standard) kinding relation [28].

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

Definition 1 (Scalar multiplication). *For a context Γ then $r \cdot \Gamma$ scales each assumption by grade r, where $r \cdot \emptyset = \emptyset$ and $r \cdot (\Gamma, x :_s A) = (r \cdot \Gamma), x :_{r \cdot s} A$.*

$$\frac{\Sigma \vdash A : \mathsf{Ty}}{\Sigma; 0 \cdot \Gamma, x :_1 A \vdash x : A}\ \textsc{Var} \qquad \frac{(x : \forall \overline{\alpha : \kappa}.A') \in D \quad \Sigma \vdash A = \mathrm{inst}(\forall \overline{\alpha : \kappa}.A')}{\Sigma; 0 \cdot \Gamma \vdash x : A}\ \textsc{Def}$$

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

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

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

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

$$\frac{\Sigma; \Gamma \vdash t : A[\mu X.A/X]}{\Sigma; \Gamma \vdash t : \mu X.A}\ \mu_1 \qquad \frac{\Sigma; \Gamma \vdash t : \mu X.A}{\Sigma; \Gamma \vdash t : A[\mu X.A/X]}\ \mu_2 \qquad \frac{\overline{\alpha : \kappa}; \emptyset \vdash t : A}{\emptyset; \emptyset \vdash t : \forall \overline{\alpha : \kappa}.A}\ \textsc{TopLevel}$$

Fig. 1: Typing rules

Top-level definitions (Def) must be present in the global definition context D, with the type scheme $\forall \overline{\alpha : \kappa}.A'$. The type A results from instantiating all of the universal variables to types via the judgment $\Sigma \vdash A = \mathrm{inst}(\forall \overline{\alpha : \kappa}.A')$ in a standard way as in Algorithm W [40]. Relatedly, the TopLevel rule types top-level definitions with polymorphic type schemes (corresponding to the generalisation rule [40]). Reading bottom up, universally quantified type variables are added to the type variable context to form the type A of the definition term t.

Abstraction (Abs) binds a variable x which is used in the body t according to grade r and thus this grade is captured onto the function arrow in the conclusion. Relatedly, application (App) scales the context Γ_2 of the argument term t_2 by the grade of the function arrow r since t_2 is used according to r in $t_1\ t_2$. To this scaled context is 'added' the context Γ_1 of the function term, via + defined:

Definition 2 (Context addition). *For contexts Γ_1, Γ_2, then $\Gamma_1 + \Gamma_2$ computes the pointwise addition using semiring addition (providing* contraction*), where:*

$$\Gamma + \emptyset = \Gamma \qquad (\Gamma_1, x :_r A) + (\Gamma_2, x :_s A) = (\Gamma_1 + \Gamma_2), x :_{r+s} A$$
$$(\Gamma_1, x :_r A) + \Gamma_2 = (\Gamma_1 + \Gamma_2), x :_r A \qquad \text{if } x \notin \mathsf{dom}(\Gamma_2)$$

For example, $(x :_1 A, y :_0 B) + x :_1 A = x :_{(1+1)} A, y :_0 B$. The operation is commutative and undefined if the type of a variable differs in two contexts.
Introduction of graded modalities is achieved via *promotion* (Pr rule) where 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 grade s provided that s *approximates* r as defined by the pre-order relation $\sqsubseteq$. This relation is occasionally lifted pointwise to contexts: we write $\Gamma \sqsubseteq \Gamma'$ to mean that Γ' over-approximates Γ, i.e., for all $(x :_r A) \in \Gamma$ then $(x :_{r'} A) \in \Gamma'$ and $r \sqsubseteq r'$.

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

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

Fig. 2: Pattern typing rules

Recursion is typed via the μ_1 rule and its inverse μ_2, in a standard way.

Introduction of data types (Con) via a constructor C of a data type $K\ \overline{A}$ (with zero or more type parameters) incurs an instantiation of its polymorphic type scheme from D. Each argument has a grade q_i. Constructors are closed, thus have only zero-use grades in the context by scaling with 0. Elimination of data types (Case) is via pattern matching. Patterns p are typed by the judgement $r \vdash\ p : A \rhd \Delta$ (Figure 2) stating that pattern p has type A and produces a context of typed binders Δ. The grade r to the left of the turnstile represents grade information arising from usage in the context generated by this pattern.

Variable patterns (PVar) produce a singleton context with $x :_r A$ of the grade r. Pattern matches on data constructors (PCon rule) may have zero or more sub-patterns ($p_1 ... p_n$), each of which is typed under the grade $q_i \cdot r$ (where q_i is the grade of corresponding argument type for the constructor, as defined in D). Additionally, we have the constraint $|K\ \overline{A}| > 1 \Rightarrow 1 \sqsubseteq r$ which witnesses the fact that if there is more than one data constructor for the data type (written $|K\ \overline{A}| > 1$), then r must approximate 1 because pattern matching on a data constructor incurs some usage since it reveals information about that constructor. [2] By contrast, pattern matching on a type with only one constructor cannot convey any information by itself and so no usage requirement is imposed. Finally, elimination of a graded modality (often called *unboxing*) takes place via the PBox rule, with syntax $[p]$. Like PCon, this rule propagates the grade information of the box pattern's type s to the enclosed sub-pattern p, yielding a context with the grades $r \cdot s$. One may observe that PBox (and by extension Pr) could be considered as special cases of PCon (and Con respectively), if we were to treat promotion as a data constructor with the type $A^r \to \Box_r A$. We however chose to keep modal introduction and elimination distinct from constructors.

Example 2. Discussed early, the natural numbers semiring with discrete ordering $(\mathbb{N}, *, 1, +, 0, \equiv)$ counts exactly how many times variables are used. We denote this semiring as $\mathbb{N}_{\equiv}$. This semiring is less useful in the presence of control-flow, e.g., for multiple branches in a **case** using variables differently. A semiring of natural number intervals [45] is more helpful here. An interval is a pair of natural numbers $\mathbb{N} \times \mathbb{N}$ written $r...s$ for lower bound $r \in \mathbb{N}$ and upper bound by $s \in \mathbb{N}$. Addition is defined pointwise with zero $0 = 0...0$ and multiplication defined as in

[2] A discussion of this additional constraint on grades is given by Hughes et al. [29]

interval arithmetic with $1 = 1...1$ and ordering $r...s \sqsubseteq r'...s' = r' \leq r \wedge s \leq s'$. This semiring allows us to write a function which performs an elimination on a coproduct (assuming $\mathsf{inl} : A^1 \to A \oplus B$, and $\mathsf{inr} : B^1 \to A \oplus B$ in D):

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

Example 3. The ! modality of linear logic can be (almost) recovered via the $\{0,1,\omega\}$ semiring where $0 \sqsubseteq \omega$ and $1 \sqsubseteq \omega$. Addition is $r + s = r$ if $s = 0$, $r + s = s$ if $r = 0$, otherwise ω. Multiplication is $r \cdot 0 = 0 \cdot r = 0$, $r \cdot \omega = \omega \cdot r = \omega$ (where $r \neq 0$), and $r \cdot 1 = 1 \cdot r = r$. This semiring expresses linear and non-linear usage, where 1 indicates linear use, 0 requires the value be discarded, and ω acts as linear logic's ! permitting arbitrary use. This is similar to Haskell's multiplicity annotations, although they have no equivalent of a 0 grade, with only `One` and `Many` grades [8]. Some additional restrictions are required on pattern typing to get exactly the behaviour of ! with respect to products [26], not considered here.

Lastly we note that the calculus enjoys admissibility of substitution [1] which is critical in type preservation proofs, and is needed for soundness of synthesis:

Lemma 1 (Admissibility of substitution). *Let $\Delta \vdash t' : A$, then: If $\Gamma, x :_r A, \Gamma' \vdash t : B$ then $\Gamma + (r \cdot \Delta) + \Gamma' \vdash [t'/x]t : B$*

4 Synthesis Calculus

Having defined the target language, we define our synthesis calculus, which uses the *additive* approach to resource management (see Section 2), with judgments:

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

That is, given an input context Γ, for goal type A we can synthesise the term t with output context Δ describing how variables were used in t. As in typing, top-level definitions and data constructors in scope are provided by a set D parameterising the system. Σ is a context of type variables, which we elide when it is simply passed inductively to the premise(s). The context Δ need not use the variables in Γ with the same grades. Instead, the relationship between synthesis and typing is given by the central soundness result, which we state up-front: that synthesised terms are typed by their goal type under their output context:

Theorem 1 (Soundness). *For all pre-ordered semirings $\mathcal{R}$:*

1. *For all contexts Γ and Δ, types A, terms t:*

$$\Sigma; \Gamma \vdash A \Rightarrow t \mid \Delta \quad \Longrightarrow \quad \Sigma; \Delta \vdash t : A$$

2. *At the top-level, for all type schemes $\forall \overline{\alpha : \kappa}.A$ and terms t then:*

$$\emptyset; \emptyset \vdash \forall \overline{\alpha : \kappa}.A \Rightarrow t \mid \emptyset \quad \Longrightarrow \quad \emptyset; \emptyset \vdash t : \forall \overline{\alpha : \kappa}.A$$

Appendix D of the additional material provides the soundness proof [28], which in part resembles a translation from sequent calculus to natural deduction, but also with the management of grades between synthesis and type checking.

The first part of soundness on its own does not guarantee that a synthesised program t is *well resourced*, i.e., the grades in Δ may not be approximated by grades in Γ. For example, for semiring $\mathbb{N}_{\equiv}$ a valid judgement is:

$$x :_2 A \vdash A \Rightarrow x \mid x :_1 A$$

i.e., for goal A, if x has type A in the context then we synthesis x as the resulting program, regardless of the grades. Such a synthesis judgement may be part of a larger derivation in which the grades eventually match due to a further subderivation, e.g., using x again and thus total usage for x is eventually 2 as prescribed by the input context. However, at the level of an individual judgement we do not guarantee that the synthesised term is well-resourced with respect to the input context. A reasonable *pruning condition* to assess whether any synthesis judgement is *potentially* well-resourced is $\exists \Delta'.(\Delta + \Delta') \sqsubseteq \Gamma$, i.e., there is some additional usage Δ' (that might come from further on in the synthesis process) that 'fills the gap' in resource use to produce $\Delta + \Delta'$ which is overapproximated by Γ. In this example, $\Delta' = x :_1 A$ would satisfy this constraint, explaining that there is some further possible single usage which will satisfy the incoming grade. However, our previous work on graded linear types showed that excessive pruning at every step becomes too costly in a general setting [27]. Instead, we apply such pruning more judiciously, only requiring that variable use is well-resourced at the point of synthesising binders. Therefore synthesised closed terms *are* always well-resourced (second part of the soundness theorem).

We next present the synthesis calculus in stages. Each type former of the core calculus (with the exception of type variables) has two corresponding synthesis rules: a right rule for introduction (labelled R) and a left rule for elimination (labelled L). We frequently apply the algorithmic reading of the judgments, where meta variables to the left of $\Rightarrow$ are inputs (i.e., context Γ and goal type A) and terms to the right of $\Rightarrow$ are outputs (i.e., the synthesised term t and the usage context Δ). Whilst we largely present the approach here in abstract terms, via the synthesis judgments, we highlight some choices made in our implementation (e.g., heuristics applied in the algorithmic version of the rules).

4.1 Core Synthesis Rules

Top-level We begin with synthesis from a type scheme goal (which is technically a separate judgment form), providing the entry-point to synthesis:

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

The universally-quantified type variables $\overline{\alpha : \kappa}$ are thus added to the type variable context of the premise (note, type variables are only equal to themselves).

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

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

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

The Var rule permits synthesis of terms which may not be well-resourced, e.g., if $r = 0$, the rule still synthesises a use of x. As discussed at this section's start, this may be locally ill-resourced, but is acceptable at the global level as we check that an assumption has been used correctly when it is bound. This reduces the number of intermediate theorems that need solving (previously shown to be expensive [27], especially since the variable rule is applied very frequently), but increases the number of paths that are ill-resourced so must be pruned later.

The use of a top-level polymorphic function is synthesised if it can be instantiated to match the goal type:

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

For example, assuming $\mathit{flip} : \forall c : \mathrm{Type}, d : \mathrm{Type}.(c \otimes d)^1 \to (d \otimes c) \in D$ then *flip* is synthesised for a goal type of $(K_1 \otimes K_2)^1 \to (K_2 \otimes K_1)$ for some type constants K_1 and K_2, via the instantiation $\emptyset \vdash (K_1 \otimes K_2)^1 \to (K_2 \otimes K_1) = \mathrm{inst}(\forall c : \mathrm{Type}, d : \mathrm{Type}.(c \otimes d)^1 \to (d \otimes c))$.

Recursion is provided by populating D with the name and type of the definition currently being synthesised for (see Section 4.2 for implementation details).

Functions Synthesis from function types is handled by the $\to_{\mathrm{R}}$rule:

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

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

Function application is synthesised from functions in the context (a left rule):

$$\frac{\begin{array}{l}\Gamma, x :_{r_1} A^q \to B, y :_{r_1} B \vdash C \Rightarrow t_1 \mid \Delta_1, x :_{s_1} A^q \to B, y :_{s_2} B \\ \Gamma, x :_{r_1} A^q \to B \vdash A \Rightarrow t_2 \mid \Delta_2, x :_{s_3} A^q \to B\end{array}}{\Gamma, x :_{r_1} A^q \to B \vdash C \Rightarrow [(x\ t_2)/y]t_1 \mid (\Delta_1 + s_2 \cdot q \cdot \Delta_2), x :_{s_2+s_1+(s_2 \cdot q \cdot s_3)} A^q \to B}\ \to_{\mathrm{L}}$$

Reading bottom up, the input context contains an assumption of function type $x :_{r_1} A^q \to B$. An application of x can be synthesised if an argument t_2 can be synthesised for the input type A (second premise). The goal type C is synthesised (first premise), under the assumption of a result of type B bound to y. In the conclusion, a term is synthesised which substitutes in t_1 the result placeholder variable y for the application $x\ t_2$.

We explain the concluding output context in two stages. Firstly, the output context Δ_1 of the first premise is added to a scaled Δ_2. Since Δ_2 are the resources used by the synthesised argument t_2, this context is scaled by q as t_2 is used according to q by x as per its type. This context is further scaled by s_2 which is the usage of the entire application $x\ t_2$ inside t_1 as given by the output grade for y in the first premise. Secondly, the output context calculates the use of x used in the application itself and potentially also by both premises (which differs from LGM's treatment of synthesis in a linear setting). Apart from application, x may be used also to synthesise the argument t_2, calculated as grade s_3 in the second premise. Thus, the application accrues $q \cdot s_3$ use. Furthermore as the result y is used according to s_2, we must further scale by s_2, obtaining $s_2 \cdot q \cdot s_3$. To this we must also add the additional usage of x in the first premise s_1 as well as the use of x in actually performing application, which is 1 scaled by s_2 to account for the usage of its result, thus obtaining the output grade for x. Following the soundness proof for this rule (Appendix D) can be instructive.

The declarative rule above does not imply an ordering of whether t_1 or t_2 is synthesised first. As a heuristic, the implementation first attempts to synthesise t_1 assuming $y :_{r_1} B$ according to the first premise to avoid possibly unnecessary work if no term can be synthesised anyway for C.

Example 4. Let $T = (A \otimes A)^{0..1} \to A$ type an assumption fst in a use of $\to_{\mathrm{L}}$:

$$\frac{\begin{array}{ll}z :_s A, \mathsf{fst} :_r T, y :_r A \vdash A \otimes A \Rightarrow (y, y) \mid z :_0 A, \mathsf{fst} :_0 T, y :_2 A \\ z :_s A, \mathsf{fst} :_r T \qquad \vdash A \Rightarrow (z, z) \mid z :_2 A, \mathsf{fst} :_0 T\end{array}}{z :_s A, \mathsf{fst} :_r T \vdash A \otimes A \Rightarrow (\mathsf{fst}\,(z, z), \mathsf{fst}\,(z, z)) \mid z :_{0+2 \cdot (0..1) \cdot 2} A, \mathsf{fst} :_{2+0+(2 \cdot (0..1) \cdot 0)} T}$$

In this instantiation of the $(\to_{\mathrm{L}})$ rule, $q = 0..1$ and $s_1 = s_3 = 0$, i.e., the function fst is not used in the subterms, and $s_2 = 2$, i.e., the result y of fst is used twice. In the conclusion then, z then has output grade $0 + 2 \cdot (0..1) \cdot 2 = 0..4$, i.e., it is used up to four times and fst has grade $2..2$, i.e., it is used twice.

Graded Modalities Graded modalities are introduced through the $\Box_{\mathrm{R}}$rule, synthesising a promotion $[t]$ for some graded modal type $\Box_r A$:

$$\frac{\Gamma \vdash A \Rightarrow t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow [t] \mid r \cdot \Delta}\ \Box_{\mathrm{R}}$$

The premise synthesises term t from A with output context Δ. In the conclusion, Δ is scaled by the grade r of the goal type since $[t]$ must use t as r requires.

Grade elimination (*unboxing*) takes place via pattern matching in **case**:

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

To eliminate an assumption x of graded modal type $\Box_q A$, we bind a fresh assumption the premise: $y :_{r \cdot q} A$. This assumption is graded with $r \cdot q$: the grade from the assumption's type multiplied by the grade of the assumption itself. As with previous elimination rules, x is rebound in the rule's premise. A term t is then synthesised resulting in the output context $\Delta, y :_{s_1} A, x :_{s_2} \Box_q A$, where s_1 and s_2 describe how y and x were used in t. The second premise ensures that the usage of y is well-resourced. The grade s_3 represents how much the usage of y inside t contributes to the overall usage of x. The constraint $s_1 \sqsubseteq s_3 \cdot q$ conveys the fact that q uses of y constitutes a single use of x, with the constraint $s_3 \cdot q \sqsubseteq r \cdot q$ ensuring that the overall usage does not exceed the binding grade. For the output context of the conclusion, we simply remove the bound y from Δ and add x, with the grade $s_2 + s_3$ representing the total usage of x in t.

Data Types The synthesis of introduction forms for data types is by the $\mathrm{C_R}$rule:

$$\frac{\begin{array}{l}(C : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\ \overline{A'}) \in D \\ \Sigma \vdash {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K\ \overline{A} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\ \overline{A'}) \\ \Sigma; \Gamma \vdash B_i \Rightarrow t_i \mid \Delta_i\end{array}}{\Sigma; \Gamma \vdash K\ \overline{A} \Rightarrow C\ t_1 \ldots t_n \mid 0 \cdot \Gamma + (q_1 \cdot \Delta_1) + \ldots + (q_n \cdot \Delta_n)}\ \mathrm{C_R}$$

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

For a goal type $K\ \overline{A}$ where K is a data type with zero or more type arguments (denoted by the vector $\overline{A}$), then a constructor term $C\ t_1 \ldots t_n$ for $K\ \overline{A}$ is synthesised. The type scheme of the constructor in D is first instantiated (similar to DEF rule), yielding a type ${B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K\ \overline{A}$. A sub-term is then synthesised for each of the constructor's arguments t_i in the third premise (which is repeated for each instantiated argument type B_i), yielding output contexts Δ_i. The output context for the rule's conclusion is obtained by performing a context addition across all the output contexts generated from the premises, where each context Δ_i is scaled by the corresponding grade q_i from the data constructor in D capturing the fact that each argument t_i is used according to q_i.

Data type elimination synthesises **case** expressions, pattern matching on each data constructor of the goal data type $K\ \overline{A}$, with various constraints on grades. In the rule, we use the least-upper bound (lub) operator $\sqcup$ on grades, which is

defined wrt. $\sqsubseteq$ and may not always be defined:

$$\frac{\begin{array}{l}(C_i : \forall \overline{\alpha : \kappa}.B_1'^{\,q_1} \to \ldots \to B_n'^{\,q_n} \to K\ \overline{A'}) \in D \qquad \Sigma \vdash K\ \overline{A} : \mathsf{Ty} \\ \Sigma \vdash B_1{}^{q_1} \to \ldots \to B_n{}^{q_n} \to K\ \overline{A} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.B_1'^{\,q_1} \to \ldots \to B_n'^{\,q_n} \to K\ \overline{A'}) \\ \Sigma; \Gamma, x :_r K\ \overline{A}, y_1^i :_{r \cdot q_1^i} B_1, \ldots, y_n^i :_{r \cdot q_1^i} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\ \overline{A}, y_1^i :_{s_1^i} B_1, \ldots, y_n^i :_{s_n^i} B_n \\ \exists s'^i_j.\, s_j^i \sqsubseteq s'^i_j \cdot q_j^i \sqsubseteq r \cdot q_j^i \qquad s_i = s'^i_1 \sqcup \ldots \sqcup s'^i_n \qquad |K\ \overline{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup \ldots \sqcup s_m\end{array}}{\Sigma; \Gamma, x :_r K\ \overline{A} \vdash B \Rightarrow \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ y_1^i \ldots y_n^i \mapsto t_i} \mid (\Delta_1 \sqcup \ldots \sqcup \Delta_m), x :_{\bigsqcup r_i + \bigsqcup s_i} K\ \overline{A}}\ \mathrm{C_L}$$

where $1 \leq i \leq m$ indexes data constructors of which there are m (i.e., $m = |K\ \overline{A}|$) and $1 \leq j \leq n$ indexes arguments of the i^{th} data constructor, thus n depends on i. The rule considers data constructors where $n > 0$ for brevity.

The relevant data constructors C_i are retrieved from the global scope D in the first premise. Each polymorphic type scheme is instantiated to a monomorphic type. The monomorphised type for each i is a function from constructor arguments $B_1 \ldots B_n$ to the applied type constructor $K\ \overline{A}$. For each C_i, we synthesise a term t_i from this result type $K\ \overline{A}$, binding the data constructor's argument types as fresh assumptions to be used in the synthesis of t_i. The grades of each argument are scaled by r. This follows the pattern typing rule for constructors; a pattern match under some grade r must bind assumptions that have the capability to be used according to r. The assumption being eliminated $x :_r K\ \overline{A}$ is also included in the premise's context (as in $\to_{\mathrm{L}}$) as we may perform additional eliminations on the current assumption subsequently.

The output context for each branch can be broken down into three parts:

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

For the concluding output context, we take the least-upper bound of the shared output contexts Δ_i of the branches. This is extended with the grade for x which requires some calculation. For each bound assumption, we generate a fresh grade variable s'^i_j which represents how that variable was used in t_i after factoring out the multiplication by q_j^i. This is done via the constraint in the third premise that $\exists s'^i_j.\, s_j^i \sqsubseteq s'^i_j \cdot q_j^i \sqsubseteq r \cdot q_j^i$. The lub of s'^i_j for all j is then taken to form a grade variable s_i which represents the total usage of x for branch i arising from the use of assumptions bound via the pattern match (i.e., not usage that arises from reusing x explicitly inside t_i). The final grade for x is then the lub of each r_i (the usages of x directly in each branch) plus the lub of each s_i (the usages of the assumptions that were bound from matching on a constructor of x).

*Example 5 (**case** synthesis).* Consider two possible synthesis results:

$$x :_r A \oplus \mathsf{Unit}, y :_s A, z :_{r \cdot q_1} A \vdash A \Rightarrow z \mid x :_0 A \oplus \mathsf{Unit}, y :_0 A, z :_1 A \qquad (1)$$

$$x :_r A \oplus \mathsf{Unit}, y :_s A \qquad \vdash A \Rightarrow y \mid x :_0 A \oplus \mathsf{Unit}, y :_1 A \qquad (2)$$

We will plug these into the rule for generating **case** as follows, where Σ has been elided and instead of using the above concrete grades we have used the abstract form of the rule (the two will be linked by equations after):

$$\frac{\begin{array}{l}\mathsf{some} : (\forall \alpha, \beta : \mathsf{Ty}.\alpha^1 \to \alpha \oplus \beta) \in D \quad \Sigma \vdash A^1 \to A \oplus \mathsf{Unit} = \mathrm{inst}(\forall \alpha, \beta : \mathsf{Ty}.\alpha^1 \to \alpha \oplus \beta) \\ \mathsf{none} : (\forall \alpha, \beta : \mathsf{Ty}.\alpha \oplus \beta) \in D \quad \Sigma \vdash A \oplus \mathsf{Unit} = \mathrm{inst}(\forall \alpha, \beta : \mathsf{Ty}.\alpha \oplus \beta) \\ (1)\ \ \Sigma; x :_r A \oplus \mathsf{Unit}, y :_s A, z :_{r \cdot q_1} A \vdash A \Rightarrow z \mid x :_0 A \oplus \mathsf{Unit}, y :_0 A, z :_{s_1} A \\ (2)\ \ \Sigma; x :_r A \oplus \mathsf{Unit}, y :_s A \quad \vdash A \Rightarrow y \mid x :_0 A \oplus \mathsf{Unit}, y :_1 A \\ \exists s_1'.\, s_1 \sqsubseteq s_1' \cdot q_1 \sqsubseteq r \cdot q_1 \quad s' = s_1' \quad |A \oplus \mathsf{Unit}| \implies 1 \sqsubseteq s_1\end{array}}{x :_r A \oplus \mathsf{Unit}, y :_s A \vdash A \Rightarrow (\mathbf{case}\ x\ \mathbf{of}\ \mathsf{some}\, z \to z; \mathsf{none} \to y) \mid x :_{(0 \sqcup 0) + s'} A \oplus \mathsf{Unit}, y :_{0 \sqcup 1} A}$$

To unify (1) and (2) with the $\mathrm{C_L}$ rule format $s_1 = 1$ and $q_1 = 1$ (from the type of inl). Applying these equalities to the existential constraint we have

$$\exists s_1'.\, 1 \sqsubseteq (s_1' \cdot 1) \sqsubseteq (r \cdot 1) \quad \implies \quad \exists s_1'.\, 1 \sqsubseteq s_1' \sqsubseteq r$$

With the natural-number intervals semiring this is satisfied by $s_1' = 1..1 = s'$ and thus in the output context x has grade 1..1 and y has grade 0..1.

Recursive Types Though μ types are equi-recursive, we define explicit synthesis rules to facilitate the implementation (Section 4.2) where depth information needs to be tracked when employing the following μ_{L} and μ_{R} rules:

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

To synthesise a recursive data structure of type $\mu X.A$, we must be able to synthesise A with $\mu X.A$ substituted for the recursion variable X in A. For example, if we wish to synthesise a list typed `List a` (where `Cons : a → List a → List a`) then when synthesising a `Cons` constructor in the μ_{R} rule, we must re-apply the μ_{R} rule to synthesise the recursive argument. Elimination of a value $\mu X.A$ in the context is via the μ_{L}, which expands the recursive type in the synthesis context.

4.2 Algorithmic Implementation

The calculus presented above serves as a starting point for our implemented synthesis algorithm in Granule. However, the rules are highly non-deterministic with regards their order in which they may be applied. For example, after applying a ($\to_{\mathrm{R}}$)-rule, we may choose to apply any of the elimination rules before applying an introduction rule for the goal type. This leads to us exploring a large number of redundant search branches which can be avoided through the application of a technique known as *focusing* [4]. Focusing is a tool from linear logic proof theory based on the idea that some rules are invertible, i.e., whenever the conclusion of the rule is derivable, then so are the premises. In other words, the order in which we apply invertible rules doesn't matter. By fixing a particular ordering on the application of invertible rules, we eliminate much of the non-determinism that arises from trying branches which differ only in the order in which invertible rules are applied. The full focusing versions of the rules from our calculus, and their

proof of soundness, can be found in Appendix E [28]. This forms the basis of our implementation with the high-level algorithm given in appendix Figure 5 as a (non-deterministic) finite state machine, which shows the ordering given to the rules under the focussing approach, starting with trying to synthesise function types before switching to eliminations rules, and so on. In standard terminology, our algorithm is 'top-down' (see, e.g., [17, 47, 23, 53]), or *goal-directed*, in which we start with a type goal and an input context and progress by gradually building the syntax tree from the empty term following the focussing-ordered rules of our calculus. This contrasts with 'bottom-up' approaches [2, 41, 44] which maintain complete programs which can be executed (tested) and combined.

Where transitions are non-deterministic in the algorithm, multiple branches are then explored in synthesis. Our implementation relies on the use of backtracking proof search, leveraging a monadic interface that provides both choice (e.g., between multiple possible synthesis options based on the goal type) and failure (e.g., when a constraint fails to hold) [33]. For every rule that generates a constraint on grades, due to binding ($\Box_{\mathrm{L}}$, $\rightarrow_{\mathrm{R}}$, C_{L}), we compile the constraints to the SMT-lib format [7] which are then discharged by the Z3 SMT solver [43]. If the constraint is invalid then we trigger the failure of this synthesis pathway, triggering backtracking via the "logic" monad [33]. A synthesised program can also be rejected by user (or due to a failing an example, see below) and synthesis then produces an alternate result (what we call a *retry*) via backtracking.

Recursive data structures present a challenge in the implementation. For example, for the list data type, how do we prevent synthesis from applying the μ_{L} rule, followed by the C_{L} rule on the `Cons` constructor ad infinitum? We resolve this issue using an *iterative deepening* approach similar to that used by Myth [48]. Programs are synthesised with elimination (and introduction) forms of constructors restricted up to a given depth. If no program is synthesised within these bounds, then the depth limits are incremented. The current depth and the depth limit are part of the state of the synthesiser. Combined with focusing this provides the basis an efficient implementation of the synthesis calculus. Furthermore, to ensure that a synthesised programs terminates, we only permit synthesis of recursive function calls which are *structurally recursive*, i.e., those which apply the recursive definition to a subterm of the function's inputs [48].

Lastly, after synthesis, a post-synthesis refactoring step runs to simplify terms and produce a more idiomatic style. For example for the k combinator type signature `k : ∀ {a b : Type} . a %1 → b %0 → a` we synthesis the term: `k = λx → λy → x`. Our refactoring procedure collects the outermost abstractions of a synthesised term and transforms them into equation-level patterns with the innermost abstraction body forming the equation body: `k x y = x`. Repeated **case** expressions are also refactored into nested pattern matches, which are part of Granule. For example, nested matching on pairs is simplified to a single **case** with nested pattern matching: `case x of (y1, y2) → case y1 of (z1, z2) → e` is refactored to `case x of ((z1, z2), y2) → e`.

Input-output Examples Further to the implementation described above, we also allow user-defined input-output examples which are checked as part of synthesis.

Our approach is deliberately naïve: we evaluate a fully synthesised candidate program against the inputs and check that the results match the corresponding outputs. Unlike sophisticated example-driven synthesis tools, the examples only influence the search procedure by backtracking on a complete program that doesn't satisfy the examples. This lets us consider the effectiveness of search based primarily around the use of grades (see Section 5). Integrating examples more tightly with the type-and-grade directed approach is further work.

Our implementation augments Granule with first-class syntax for specifying input-output examples, both as a feature for aiding synthesis but also for aiding documentation that is type checked (and therefore more likely to stay consistent with a code base as it evolves). Synthesis specifications are written in Granule directly above a program hole (written using `?`) using the `spec` keyword. The input-output examples are then listed per-line. For example, one of benchmark programs (Section 5) for the length of a list is specified as:

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

Any synthesised definition must then behave according to this example.

In a `spec` block, a user can also specify the names of functions in scope which are to be taken as the available definitions (set D in the formal specification). For example, line 4 above specifies that `length` can be used here (i.e., recursively).

5 Evaluation

In evaluating our approach and tool, we made the following hypotheses:

H1. (**Expressivity; less consultation**) The use of grades in synthesis results in a synthesised program that is more likely to have the behaviour desired by the user; the user needs to request fewer alternate synthesised results (*retries*) and thus is consulted less in order to arrive at the desired program.

H2. (**Expressivity; fewer examples**) Grade-and-type directed synthesis requires fewer input-output examples to arrive at the desired program compare with a purely type-driven approach.

H3. (**Performance; more pruning**) The ability to prune resource-violating candidate programs from the search tree leads to a synthesised program being found more quickly when synthesised from a graded type compared with the same type but without grades (purely type-driven approach).

5.1 Methodology

To evaluate our approach, we collected a suite of benchmarks comprising graded type signatures for common transformations on structures such as lists, streams, booleans, option ('maybe') types, unary natural numbers, and binary trees. A

representative sample of benchmarks from the Myth synthesis tool [47] are included alongside a variety of other programs one might write in a graded setting. Benchmarks are categorised based on the main data type, with an additional miscellaneous category. Appendix C lists type schemes for all benchmarks [28]. To compare, in various ways, our grade-and-type-directed synthesis to traditional type-directed synthesis, each benchmark signature is also "de-graded" by replacing all grades in the type with `Any` which is the only element of the singleton `Cartesian` semiring in Granule. When synthesising in this semiring, we can forgo discharging grade constraints in the SMT solver entirely. Thus, synthesis for Cartesian grades degenerates to type-directed synthesis following our rules.

To assess hypothesis 1 (grade-and-type directed leads to less consultation / more likely to synthesise the intended program) we perform grade-and-type directed synthesis on each benchmark problem and type-directed synthesis on the corresponding de-graded version. For the de-graded versions, we record the number of retries N needed to arrive at a well-resourced answer by type checking the output programs against the original graded type signature, retrying if the program is not well-typed (essentially, not well-resourced). This checks whether a program is 'as intended' without requiring input from a user. In each case, we also compared whether the resulting programs from synthesis via graded-and-type directed vs. type-directed with retries (on non-we were equivalent.

To assess hypothesis 2 (graded-and-type directed requires fewer examples than type-directed), we run the de-graded (Cartesian) synthesis with the smallest set of examples which leads to the model program being synthesised (without any retries). To compare across approaches to the state-of-the-art type-directed approach, we also run a separate set of experiments comparing the minimal number of examples required to synthesise in Granule (with grades) vs. Myth.

To assess hypothesis 3 (grade-and-type-directed faster than type-directed) we compare performance in the graded setting to the de-graded Cartesian setting. Comparing our tool for speed against another type-directed (but not graded-directed) synthesis tool such as Myth is likely to be largely uninformative due to differences in implementation (engineering artefacts) obscuring meaningful comparison. Thus, we instead compare timings for the graded and de-graded approach within Granule. This normalises implementation artefacts as the two approaches vary only in the use of SMT solving to prune ill-resourced programs (in the graded approach). We also record the number of search paths taken (over all retries) to assess the level of pruning in the graded vs de-graded case.

We ran our synthesis tool on each benchmark for both the graded type and the de-graded Cartesian case, computing the mean after 10 trials for timing data. Benchmarking was carried out using version 4.12.1 of Z3 [43] on an M1 MacBook Air with 16 GB of RAM. A timeout limit of 10 seconds was set for synthesis.

5.2 Results and Analysis

Table 1 records the results comparing grade-and-type synthesis vs. the Cartesian (de-graded) type-directed synthesis. The left column gives the benchmark name, number of top-level definitions in scope that can be used as components (size

of the synthesis context) labelled Ctxt, and the minimum number of examples needed (#/Exs) to synthesise the Graded and Cartesian programs. In the Cartesian setting, where grade information is not available, if we forgo type-checking a candidate program against the original graded type then additional input-output examples are required to provide a strong enough specification such that the correct program is synthesised (see H3). The number of additional examples is given in parentheses for those benchmarks which required these additional examples to synthesise a program in the Cartesian setting.

Each subsequent results column records: whether a program was synthesised successfully ✓ or not × (due to timeout or no solution found), the mean synthesis time (μT) or if timeout occurred, and the number branching paths (Paths) explored in the synthesis search space.

The first results column (Graded) contains the results for graded synthesis. The second results column (Cartesian + Graded type-check) contains the results for synthesising in the Cartesian (de-graded) setting, using the same examples set as the Graded column, and recording the number of retries (consultations of the type-checker at the end) N needed to reach a well-resourced program. In all cases, the resulting program in the Cartesian case was equivalent to that generated by the graded synthesis, none of which needed any retries (i.e., implicitly $N = 0$ for graded synthesis, i.e., no retries are needed). H1 is confirmed by the fact that N is greater than 0 in 29 out of 46 benchmarks (60%), i.e., the Cartesian case does not synthesis the correct program first time and needs multiple retries to reach a well-resource program, with a mean of 19.60 retries and a median of 4 retries.

For each row, we highlight the column which synthesised a result the fastest in blue. In 17 of the 46 benchmarks (37%) the graded approach out-performed non-graded synthesis. This contradicts hypothesis 3 somewhat: whilst type-directed synthesis often requires multiple retries (versus no retries for graded) it still outperforms graded synthesis. This is due to the cost of SMT solving which must compile a first-order theorem on grades into the SMT-lib file format, start Z3, and then run the solver. Considerable amounts of system overhead are incurred in this procedure. A more efficient implementation calling Z3 directly (via a dynamic library call) may give more favourable results here. However, H3 is still somewhat supported: the cases in which the graded does outperform the Cartesian are those which involve considerable complexity in their use of grades, such as `stutter`, `inc`, and `bind` for lists, and `sum` for both lists and trees. In each case, the Cartesian column is significantly slower, even timing out for `stutter`; this shows the power of the graded approach. Furthermore, we highlight the column with the smallest number of synthesis paths explored in yellow, observing that the number of paths in the graded case is always the same or less than that those in the Cartesian+graded type check case (apart from Tree stutter). The paths explored are the sometimes the same between Graded and Cartesian synthesis because we use backtracking search even in the Cartesian case where, if an output program fails to type check against the graded type, the search backtracks rather than starting from the beginning. This leads to an equal number of paths in the graded case when solving occurred only at a top-level abstraction. How-

ever, paths explored are fewer in the graded case when solving occurs at other binders, e.g., in **case** and unboxing.

Confirming H2, the de-graded setting without graded type checking requires more examples to synthesise the same program as the graded in 20 out of 46 (43%) cases. In these cases, an average of 1.25 additional examples are required. To further interrogate H2, we compare the number of examples required by

				Graded		Cartesian + Graded type-check			
	Problem	Ctxt	#/Exs.		μT (ms)	Paths		μT (ms)	N	Paths
List	append	0	0 (+1)	✓	115.35 (5.13)	130	✓	105.24 (0.36)	8	130
	concat	1	0 (+3)	✓	1104.76 (1.60)	1354	✓	615.29 (1.43)	12	1354
	empty	0	0	✓	5.31 (0.02)	17	✓	1.20 (0.01)	0	17
	snoc	1	1	✓	2137.28 (2.14)	2204	✓	1094.03 (4.75)	8	2278
	drop	1	1	✓	1185.03 (2.53)	1634	✓	445.95 (1.71)	8	1907
	flatten	2	1	✓	1369.90 (2.60)	482	✓	527.64 (1.04)	8	482
	bind	2	0 (+2)	✓	62.20 (0.21)	129	✓	622.84 (0.95)	18	427
	return	0	0 (+1)	✓	19.71 (0.18)	49	✓	22.00 (0.08)	4	49
	inc	1	1	✓	708.23 (0.69)	879	✓	2835.53 (7.69)	24	1664
	head	0	1	✓	68.23 (0.53)	34	✓	20.78 (0.10)	4	34
	tail	0	1	✓	84.23 (0.20)	33	✓	38.59 (0.06)	8	33
	last	1	1 (+1)	✓	1298.52 (1.17)	593	✓	410.60 (6.25)	4	684
	length	1	1	✓	464.12 (0.90)	251	✓	127.91 (0.58)	4	251
	map	1	0 (+1)	✓	550.10 (0.61)	3075	✓	249.42 (0.73)	4	3075
	replicate5	0	0 (+1)	✓	372.23 (0.70)	1295	✓	435.78 (1.06)	4	1295
	replicate10	0	0 (+1)	✓	2241.87 (4.74)	10773	✓	2898.93 (1.47)	4	10773
	replicateN	1	1	✓	593.86 (1.68)	772	✓	108.98 (0.65)	4	772
	stutter	1	0	✓	1325.36 (1.77)	1792	×	Timeout	-	-
	sum	2	1 (+1)	✓	84.09 (0.25)	208	✓	3236.74 (0.87)	192	3623
Stream	build	0	0 (+1)	✓	61.27 (0.45)	75	✓	84.44 (0.49)	4	75
	map	1	0 (+1)	✓	351.93 (0.91)	1363	✓	153.01 (0.37)	0	1363
	take1	0	0 (+1)	✓	34.02 (0.23)	22	✓	19.32 (0.05)	0	22
	take2	0	0 (+1)	✓	110.18 (0.31)	204	✓	89.10 (0.18)	0	208
	take3	0	0 (+1)	✓	915.39 (1.42)	1139	✓	631.47 (1.14)	0	1172
Bool	neg	0	2	✓	209.09 (0.31)	42	✓	168.37 (0.56)	0	42
	and	0	4	✓	3129.30 (2.82)	786	✓	7069.14 (15.91)	0	2153
	impl	0	4	✓	1735.09 (4.31)	484	✓	3000.48 (4.65)	0	1214
	or	0	4	✓	1213.86 (1.02)	374	✓	2867.74 (3.52)	0	1203
	xor	0	4	✓	2865.79 (4.33)	736	✓	7251.38 (32.06)	0	2229
Maybe	bind	0	0 (+1)	✓	159.87 (0.52)	237	✓	55.33 (0.33)	0	237
	fromMaybe	0	0 (+2)	✓	54.27 (0.35)	18	✓	11.58 (0.10)	0	18
	return	0	0	✓	9.89 (0.02)	17	✓	11.49 (0.04)	4	17
	isJust	0	2	✓	69.33 (0.17)	48	✓	22.07 (0.09)	0	48
	isNothing	0	2	✓	102.42 (0.32)	49	✓	31.89 (0.22)	0	49
	map	0	0 (+1)	✓	54.90 (0.22)	120	✓	22.01 (0.10)	0	120
	mplus	0	1	✓	319.64 (0.47)	318	✓	70.98 (0.05)	0	318
Nat	isEven	1	2	✓	1027.79 (1.28)	466	✓	313.77 (0.92)	8	468
	pred	0	1	✓	46.20 (0.18)	33	✓	48.04 (0.13)	8	33
	succ	0	1	✓	115.16 (0.91)	76	✓	156.02 (0.50)	8	76
	sum	1	1 (+2)	✓	1582.23 (3.60)	751	✓	734.38 (1.41)	12	751
Tree	map	1	0 (+1)	✓	1168.60 (1.21)	4259	✓	525.47 (1.31)	4	4259
	stutter	1	0 (+1)	✓	693.44 (1.21)	832	✓	219.91 (1.02)	4	674
	sum	2	3	✓	1477.83 (1.28)	3230	✓	3532.24 (7.19)	192	3623
Misc	compose	0	0	✓	40.27 (0.08)	38	✓	14.53 (0.09)	2	38
	copy	0	0	✓	5.24 (0.04)	21	✓	6.16 (0.10)	2	21
	push	0	0	✓	26.66 (0.18)	45	✓	14.23 (0.13)	2	45

Table 1: Results. μT in *ms* to 2 d.p. with standard sample error in brackets

Problem		Granule #/Exs	Myth #/Exs	Smyth #/Exs
List	append	0	6	4
	concat	1	6	3
	snoc	1	8	3
	drop	1	13	5
	inc	1	4	2
	head	1	3	2
	tail	1	3	2
	last	1	6	4
	length	1	3	3
	map	0	8	4
List	stutter	0	3	2
	sum	1	3	2
Bool	neg	2	2	2
	and	4	4	3
	impl	4	4	3
	or	4	4	3
	xor	4	4	4
Nat	isEven	2	4	3
	add	1	3	2
	pred	1	3	2
Tree	map	0	7	4

Table 2: Number of examples needed for synthesis, Granule vs. Myth vs. Smyth

Granule (using grades) against the Myth synthesis tool (based on pruning by examples) [47], and the more advanced assertion-based Smyth [36]. We consider the subset of our benchmarks drawn from Myth. Table 2 shows the minimum number of input-output examples needed to synthesise the correct program in Granule, Myth, and Smyth. For all cases, Granule required the same or fewer examples than Myth to synthesis the desired program, requiring fewer examples in 16 out of 21 cases. The disparity in the number of examples required is quite significant in some cases: with 13 examples required by Myth to synthesise *concat* but only 1 example for Granule. Overall, Smyth needed the same or fewer examples than Myth. Granule needed the same or fewer examples than Smyth in 18 out of 21 cases, but in the other 3 cases (and, impl, or) Smyth required 1 fewer example. Overall, the lower number of examples needed in our approach shows the pruning power of grades in synthesis, confirming H2.

We briefly examine one of the more complex benchmarks which uses almost all of our synthesis rules in one program. The `stutter` case (List class) is specified:

```
1 stutter : ∀ a . List (a [2]) %1..∞ → List a
2 spec stutter
```

Its input is a list of elements graded by `2`, i.e., must be used twice. The argument list itself must be used at least once but possibly infinitely, suggesting that some recursion will be necessary. This is further emphasised by the `spec`, which states we can use `stutter` itself inside the function. Without grades, synthesis times out. Graded synthesise produces the following in 1325ms (∼1.3 seconds):

```
1 stutter Nil = Nil;
2 stutter (Cons [u] z) = (Cons u) ((Cons u) (stutter z))
```

6 Synthesis of Linear Haskell Programs

As part of a growing trend of resourceful types being added to more mainstream languages, Haskell has introduced support for linear types as of GHC 9, using an underlying graded type system which can be enabled as a language extension [8] (called `LinearTypes`). This system is closely related to the calculus here but limited to one semiring. This however presents an opportunity to leverage our

tool to synthesise (linear) Haskell programs. Like Granule, grades in Haskell can be expressed as "multiplicities" on function types: `a %r -> b`. The multiplicity r can be either 1 or ω (or polymorphic), with 1 denoting linear usage (also written as `'One`) and ω (`'Many`) for unrestricted use. Similarly, Granule can model linear types using the 0-1-ω semiring (Example 1) [26]. Synthesising Linear Haskell programs then simply becomes a task of parsing a Haskell type into a Granule equivalent, synthesising a term from it, and compiling the synthesised term back to Haskell (which has similar syntax to Granule anyway).

Our implementation includes a prototype synthesis tool using this approach. A synthesis problem takes the form of a Linear Haskell program with a hole, e.g.

```
1 {-# LANGUAGE LinearTypes #-}
2 swap :: (a, b) %One -> (b, a)
3 swap = _
```

We invoke the synthesis tool with `gr --linear-haskell swap.hs` which produces:

```
1 swap (z, y) = (y, z)
```

Users may make use of lists, tuples, `Maybe` and `Either` data types from Haskell's prelude, as well as user-defined ADTs. Further integration of the tool, as well as support for additional Haskell features such as GADTs is left as future work.

7 Discussion

Comparison with prior work Previously, LGM targeted the linear λ-calculus with graded modalities [27]. In this paper, we instead considered a fully-graded ('graded base') calculus with no linearity: all assumptions are graded and subsequently there is a graded function arrow (not present in the 'linear base' style). This graded calculus matches practical implementations of graded types seen in Idris 2 and Haskell. Furthermore, a key contribution beyond LGM is the handling of recursion, general user-defined (recursive) ADTs, and polymorphism. Due to the pervasive grading, the majority of the synthesis rules are considerably different to LGM. For example, LGM's synthesis of functions is linear, and thus need not handle the complexity of grading (*cf.* $\rightarrow_{\mathrm{L}}$ on p. 13):

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1; \Delta_1, x_2 : B \qquad \Gamma \vdash A \Rightarrow^{+} t_2; \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{+} [(x_1 t_2)/x_2]t_1; (\Delta_1 + \Delta_2), x_1 : A \multimap B} L \multimap^{+} [27]$$

As above, in the linear setting of LGM, many of the constraints and grades handled in this paper are essentially specialised away as equal to 1, with only linear products and coproducts considered. Since grading is potentially more permissive than linearity, elimination rules in our synthesis calculus must also make available an eliminated variable for re-use in every premise, which was not needed in LGM. Furthermore, the power of this paper's **case** rule means there are simple, non-recursive terms we can synthesise which LGM cannot. In particular, synthesis of programs which perform "deep" pattern matching over a graded data structure are not possible in LGM. For example, LGM's approach cannot

synthesise a term for $\Box_{0..1}(\alpha, (\alpha, \beta)) \multimap \beta$ as it cannot propagate information from one **case** to another to inherit the grade 0..1 on the pair's components. However, here we can synthesise (in just a few steps, plus refactoring):

```
deep : ∀ a b : Type . (a, (a, b)) %0..1 → b
deep [(_, (_, y))] = y                    -- y inherits grade 0..1
```

Thus, not only does our approach consider a different mode of grading, as well as extending to arbitrary recursive ADTs and recursive functions, it is also more expressive in the interaction between data types and grades.

LGM introduced *additive* and *subtractive* resource management schemes (summarised and re-contextualised in Section 2). Comparative evaluation of LGM showed that constraints from the subtractive approach are typically larger, more complex, and discharged more frequently than in additive synthesis. We concluded that subtractive only ever outperformed additive on purely linear types. Coupled with the fact that the subtractive approach has limitations in the presence of polymorphic grades, we thus adopted the additive scheme, especially in light of us considering more complex programs. Our evaluation of LGM did not given any evidence justifying use of grades for synthesis compared to just using types. Here, we showed that grading significantly reduces the number of paths explored and examples needed when compared with purely type-directed approaches, including in comparison with MYTH [47].

Other Related Work Beyond MYTH, other recent work has extended type-and-example-directed synthesis approaches. SMYTH constrains the search space further by augmenting types with assertions (called 'sketches') to guide synthesis [36]. This techniques involves employing more evaluation during synthesis to generated intermediate input-output examples to prune the search space. They evaluate on a subset of MYTH benchmarks (somewhat similar to our own method here). Whilst we compared our approach (with graded + types + examples considered at the end) to MYTH (with types + examples integrated) to show that grading reduces the number of examples, comparing with the assertion-based approach in SMYTH is further work. Another recent work, BURST, also leverages the MYTH benchmark, but using a 'bottom-up' technique [41] (in contrast to our top-down approach, Section 4.2). The bottom-up approach synthesises a sequence of complete programs which can be refined and tested under an 'angelic semantics'. Whether a bottom-up grade-directed approach could lead to performance improvements is an open question.

Whilst we considered resourceful programming via graded types, other notions of resourceful typing exist, including 'ownership' (e.g., Rust [31]) and related 'uniqueness' (e.g., Clean [51]). Recently, Fiala et al. synthesised Rust programs from a custom program logic *Synthetic Ownership Logic* that integrates a typed approach to Rust ownership with functional specifications, allowing synthesis to follow a deductive approach [16]. There is some philosophical overlap in the resourceful ideas in their approach and ours. Drawing a closer correspondence between Rust-style ownership and grading, to perhaps leverage our resourceful approach to synthesis, is future work. Notably, Marshall and Orchard show that

uniqueness types can be implemented as an extension of a linear type theory with a non-linearity modality and uniqueness modality [38]. Further work could adapt our approach to this setting to provide synthesis for uniqueness types as a precursor to the full ownership and borrowing system of Rust.

The dependently-typed language Idris provides automated proof search as part of its implementation [10]. In Idris 2, the core type theory is based on a graded type theory [5, 39] with 0-1-ω semiring (Example 1) and with proof synthesis extended to utilise these grades [11]. This approach has some relation to ours, but in a limited single-semiring setting and restricted in how grades can be leveraged. Our approach is readily applicable to Idris, which is future work.

Conclusion Our work is grounded in the philosophy of type-driven development where the user thinks about the expected behaviour or constraints of a program first, writing the type as a specification. Synthesis is not necessarily about having complicated programs generated but is often about generating straightforward programs to save effort. This is the gain provided by type-directed synthesis in existing languages such as Agda [9] and Idris [10]. Our technique augments this, such that boilerplate code and simple algorithms can be automatically generated, freeing the developer to focus on other parts of a program.

A next step is to incorporate GADTs (Generalised ADTs), i.e., indexed types, into synthesis. Granule provides support for user-defined GADTs, and the interaction between grades and type indices is a key contributor to its expressive power [45]. For example, consider a function that replicates a value a number of times to create a list, typed `rep : ∀ {t : Type} . Int → t % 0..∞ → List t`. Given a standard indexed type of natural numbers `N (n : Nat)` and sized-indexed vectors `Vec (n : Nat) (t : Type)`, a more precise specification can be given as `∀ {n : Nat, t : Type} . N n → t % n → Vec n t` for which the search space could be more effectively pruned by including type indices in synthesis.

We intend to pursue further improvements to our tool to reduce the overhead of SMT solving, integrate examples into the search algorithm itself in the style of Myth [47] and Leon [34], as well as considering possible semiring-dependent optimisations that may be applicable. Another further work is prove completeness of our synthesis calculus which we believe this holds.

With the rise in Large Language Models showing their power at program synthesis [6, 30] the deductive approach still has value, providing correct-by-construction synthesis from specification rather than predicting programs which may violate fine-grained type constraints, e.g., from grades. Future work, and a general challenge for the deductive synthesis community, is to combine the two approaches with the logical engine of the deductive approach guiding prediction.

Data-Availability An artefact supporting the results of this work is available at `http://zenodo.org/records/10511509`.

Acknowledgements Thank you to Frank Pfenning and the anonymous reviewers for their helpful comments, and to Ben Orchard for his help in preparing this work's artefact. This work was supported by an EPSRC grant EP/T013516/1.

Bibliography

[1] Abel, A., Bernardy, J.: A unified view of modalities in type systems. Proc. ACM Program. Lang. **4**(ICFP), 90:1–90:28 (2020). https://doi.org/10.1145/3408972, https://doi.org/10.1145/3408972

[2] Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. pp. 934–950 (2013). https://doi.org/10.1007/978-3-642-39799-8_67, https://doi.org/10.1007/978-3-642-39799-8_67

[3] Allais, G.: 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] Andreoli, J.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation **2**(3), 297–347 (06 1992). https://doi.org/10.1093/logcom/2.3.297

[5] Atkey, R.: 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. pp. 56–65 (2018). https://doi.org/10.1145/3209108.3209189, https://doi.org/10.1145/3209108.3209189

[6] Austin, J., Odena, A., Nye, M.I., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C.J., Terry, M., Le, Q.V., Sutton, C.: Program synthesis with large language models. CoRR **abs/2108.07732** (2021), https://arxiv.org/abs/2108.07732

[7] Barrett, C., Stump, A., Tinelli, C., et al.: The smt-lib standard: Version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK). vol. 13, p. 14 (2010)

[8] Bernardy, J., Boespflug, M., Newton, R.R., Jones, S.P., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. Proc. ACM Program. Lang. **2**(POPL), 5:1–5:29 (2018). https://doi.org/10.1145/3158093, https://doi.org/10.1145/3158093

[9] Bove, A., Dybjer, P., Norell, U.: A brief overview of agda–a functional language with dependent types. In: Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22. pp. 73–78. Springer (2009)

[10] Brady, E.C.: Idris, a general-purpose dependently typed programming language: Design and implementation. J. Funct. Program. **23**(5), 552–593 (2013). https://doi.org/10.1017/S095679681300018X, https://doi.org/10.1017/S095679681300018X

[11] Brady, E.C.: Idris 2: Quantitative type theory in practice **194**, 9:1–9:26 (2021). https://doi.org/10.4230/LIPICS.ECOOP.2021.9, https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

[12] Brunel, A., Gaboardi, M., Mazza, D., Zdancewic, S.: A core quantitative coeffect calculus. In: Shao, Z. (ed.) 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. Lecture Notes in Computer Science, vol. 8410, pp. 351–370. Springer (2014). https://doi.org/10.1007/978-3-642-54833-8_19

[13] Cervesato, I., Hodas, J.S., Pfenning, F.: Efficient resource management for linear logic proof search. Theoretical Computer Science **232**(1), 133 – 163 (2000). https://doi.org/https://doi.org/10.1016/S0304-3975(99)00173-5

[14] Choudhury, P., III, H.E., Eisenberg, R.A., Weirich, S.: A graded dependent type system with a usage-aware semantics. Proc. ACM Program. Lang. **5**(POPL), 1–32 (2021). https://doi.org/10.1145/3434331, https://doi.org/10.1145/3434331

[15] Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. SIGPLAN Not. **50**(6), 229–239 (jun 2015). https://doi.org/10.1145/2813885.2737977, https://doi.org/10.1145/2813885.2737977

[16] Fiala, J., Itzhaky, S., Müller, P., Polikarpova, N., Sergey, I.: Leveraging rust types for program synthesis. Proceedings of the ACM on Programming Languages **7**(PLDI), 1414–1437 (2023)

[17] Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. ACM SIGPLAN Notices **51**(1), 802–815 (2016)

[18] Gaboardi, M., Katsumata, S.y., Orchard, D., Breuvart, F., Uustalu, T.: Combining Effects and Coeffects via Grading. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. p. 476–489. ICFP 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2951913.2951939, https://doi.org/10.1145/2951913.2951939

[19] Ghica, D.R., Smith, A.I.: Bounded linear types in a resource semiring. In: Shao, Z. (ed.) Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014. Lecture Notes in Computer Science, vol. 8410, pp. 331–350. Springer (2014). https://doi.org/10.1007/978-3-642-54833-8_18

[20] Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1 – 101 (1987). https://doi.org/https://doi.org/10.1016/0304-3975(87)90045-4

[21] Girard, J.Y., Scedrov, A., Scott, P.J.: Bounded linear logic: a modular approach to polynomial-time computability. Theoretical computer science **97**(1), 1–66 (1992)

[22] Green, C.: Application of theorem proving to problem solving. In: Proceedings of the 1st International Joint Conference on Artificial Intelligence. p. 219–239. IJCAI'69, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1969)

[23] Gulwani, S.: Automating string processing in spreadsheets using input-output examples. ACM Sigplan Notices **46**(1), 317–330 (2011)

[24] Harland, J., Pym, D.J.: Resource-distribution via boolean constraints. CoRR **cs.LO/0012018** (2000), https://arxiv.org/abs/cs/0012018

[25] Hodas, J., Miller, D.: Logic Programming in a Fragment of Intuitionistic Linear Logic. Information and Computation **110**(2), 327 – 365 (1994). https://doi.org/https://doi.org/10.1006/inco.1994.1036

[26] Hughes, J., Marshall, D., Wood, J., Orchard, D.: Linear Exponentials as Graded Modal Types. In: 5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021). Rome (virtual), Italy (Jun 2021), https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271465

[27] Hughes, J., Orchard, D.: Resourceful program synthesis from graded linear types. In: Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings. pp. 151–170 (2020). https://doi.org/10.1007/978-3-030-68446-4_8, https://doi.org/10.1007/978-3-030-68446-4_8

[28] Hughes, J., Orchard, D.: Program Synthesis from Graded Types (Additional Material) (Jan 2024). https://doi.org/10.5281/zenodo.10594378, https://zenodo.org/records/10594378

[29] Hughes, J., Vollmer, M., Orchard, D.: Deriving distributive laws for graded linear types. In: Lago, U.D., de Paiva, V. (eds.) Proceedings Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity&TLLA@IJCAR-FSCD 2020, Online, 29-30 June 2020. EPTCS, vol. 353, pp. 109–131 (2020). https://doi.org/10.4204/EPTCS.353.6, https://doi.org/10.4204/EPTCS.353.6

[30] Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., Sharma, R.: Jigsaw: Large language models meet program synthesis (2021)

[31] Jung, R., Dang, H.H., Kang, J., Dreyer, D.: Stacked borrows: An aliasing model for rust. Proceedings of the ACM on Programming Languages **4**(POPL), 1–32 (2019)

[32] Katsumata, S.: Parametric effect monads and semantics of effect systems. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 633–646. ACM (2014). https://doi.org/10.1145/2535838.2535846

[33] Kiselyov, O., Shan, C.c., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers: (functional pearl). SIGPLAN Not. **40**(9), 192–203 (Sep 2005). https://doi.org/10.1145/1090189.1086390

[34] Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. p. 407–426. OOPSLA '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2509136.2509555, https://doi.org/10.1145/2509136.2509555

[35] Knoth, T., Wang, D., Polikarpova, N., Hoffmann, J.: Resource-Guided Program Synthesis. CoRR **abs/1904.07415** (2019), http://arxiv.org/abs/1904.07415

[36] Lubin, J., Collins, N., Omar, C., Chugh, R.: Program sketching with live bidirectional evaluation. Proceedings of the ACM on Programming Languages **4**(ICFP), 1–29 (2020)

[37] Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**, 90–121 (01 1980). https://doi.org/10.1145/357084.357090

[38] Marshall, D., Vollmer, M., Orchard, D.: Linearity and uniqueness: An entente cordiale. In: Sergey, I. (ed.) Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13240, pp. 346–375. Springer (2022). https://doi.org/10.1007/978-3-030-99336-8_13, https://doi.org/10.1007/978-3-030-99336-8_13

[39] McBride, C.: I Got Plenty o' Nuttin', pp. 207–233. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-30936-1_12

[40] Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences **17**(3), 348–375 (1978)

[41] Miltner, A., Nuñez, A.T., Brendel, A., Chaudhuri, S., Dillig, I.: Bottom-up synthesis of recursive functional programs using angelic execution. Proceedings of the ACM on Programming Languages **6**(POPL), 1–29 (2022)

[42] Moon, B., III, H.E., Orchard, D.: Graded Modal Dependent Type Theory. In: Yoshida, N. (ed.) 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. Lecture Notes in Computer Science, vol. 12648, pp. 462–490. Springer (2021). https://doi.org/10.1007/978-3-030-72019-3_17, https://doi.org/10.1007/978-3-030-72019-3_17

[43] de Moura, L., Bjørner, N.: Z3: an efficient smt solver. vol. 4963, pp. 337–340 (04 2008)

[44] Odena, A., Shi, K., Bieber, D., Singh, R., Sutton, C., Dai, H.: Bustle: Bottom-up program synthesis through learning-guided exploration. arXiv preprint arXiv:2007.14381 (2020)

[45] Orchard, D., Liepelt, V., III, H.E.: Quantitative program reasoning with graded modal types. PACMPL **3**(ICFP), 110:1–110:30 (2019). https://doi.org/10.1145/3341714

[46] Orchard, D.A., Petricek, T., Mycroft, A.: The semantic marriage of monads and effects. CoRR **abs/1401.5391** (2014), http://arxiv.org/abs/1401.5391

[47] Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. SIGPLAN Not. **50**(6), 619–630 (Jun 2015). https://doi.org/10.1145/2813885.2738007

[48] Osera, P.M.S.: Program synthesis with types (2015)

[49] Petricek, T., Orchard, D., Mycroft, A.: Coeffects: a calculus of context-dependent computation. In: Proceedings of the 19th ACM SIGPLAN international conference on Functional programming. pp. 123–135. ACM (2014). https://doi.org/10.1145/2692915.2628160

[50] Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 522–538. PLDI '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908093, https://doi.org/10.1145/2908080.2908093

[51] Smetsers, S., Barendsen, E., van Eekelen, M., Plasmeijer, R.: Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In: Schneider, H.J., Ehrig, H. (eds.) Graph Transformations in Computer Science. pp. 358–379. Springer Berlin Heidelberg, Berlin, Heidelberg (1994). https://doi.org/10.1007/3-540-57787-4_23

[52] Smith, C., Albarghouthi, A.: Synthesizing differentially private programs. Proc. ACM Program. Lang. **3**(ICFP) (Jul 2019). https://doi.org/10.1145/3341698

[53] Yuan, Y., Radhakrishna, A., Samanta, R.: Trace-guided inductive synthesis of recursive functional programs. Proceedings of the ACM on Programming Languages **7**(PLDI), 860–883 (2023)

[54] Zalakain, U., Dardha, O.: Pi with leftovers: a mechanisation in Agda. arXiv preprint arXiv:2005.05902 (2020)