Program Synthesis from Graded Types

Anonymous authors

No Institute Given

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*) on data, a programmer can express structural or semantic properties of their program at the type level. Such systems have become increasingly popular in recent years, mainly for the expressive power that they offer to programmers—judicious use of grades in type specifications significantly reduces 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 language with 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. Our type-and-graded-directed approach is demonstrated in the research language Granule but we also adapt it for synthesising Haskell programs using GHC's linear types extension.

1 Introduction

Type-directed program synthesis is a technique for synthesising programs from a user-provided type specification. The technique has a long history intertwined with proof search, thanks to the Curry-Howard correspondence (Manna and Waldinger, 1980; Green, 1969). 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 treatment of program synthesis as a form of proof search in logic: given a type A we want to find a program term t which inhabits A. We can express this in terms of a synthesis *judgement* which acts as a kind of inversion of typing or proof rules:

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

meaning that the term t can be synthesised for the goal type A under a context of assumptions Γ. A calculus of synthesis *rules* for a 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:

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

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 assumption choices to synthesise a pair. Consider the following partial program with a *hole* (marked ?) specifying a position to perform program synthesis:

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

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

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

Even in this simple setting, the number of possibilities starts to become unwieldy: there are nine (3^2) possible candidate programs based on combinations of x, y and z. Ideally, we would like some way of constraining the number of choices that are required by the synthesis algorithm. Many systems achieve this by allowing the user to specify additional information about their desired program behaviour. For example, recent work has extended type-directed synthesis to refinement types (Polikarpova et al., 2016), cost specifications (Knoth et al., 2019), differential privacy (Smith and Albarghouthi, 2019), example-guided synthesis (Feser et al., 2015; Albarghouthi et al., 2013) or examples integrated with types (Frankle et al., 2016; Osera and Zdancewic, 2015), and ownership information (Fiala et al., 2023). The general idea is that, with more information, whether that be richer types, additional examples, or behavioural specifications, the proof search / program synthesis process can be pruned and refined.

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, with arbitrary copying and discarding disallowed (Girard, 1987). Non-linearity is expressed through the ! modal operator (the *exponential modality*). This gives

a binary view—a value may either be used exactly once or in a completely unconstrained way. Bounded Linear Logic (BLL) refines this view, replacing ! with a family of indexed modal operators where the index provides an upper bound on usage (Girard et al., 1992), e.g., $!_{\leq 4}A$ represents a value A which may be used up to 4 times. In recent years, various works have generalised BLL, resulting in *graded* type systems in which these indices are drawn from an arbitrary pre-ordered semiring (Brunel et al., 2014; Ghica and Smith, 2014; Petricek et al., 2014; Abel and Bernardy, 2020; Choudhury et al., 2021; Atkey, 2018; McBride, 2016). This allows numerous program properties to be tracked and enforced statically, including various kinds of reuse, privacy and confidentiality, and capabilities. Such systems are increasingly popular and form the basis of Linear Haskell (Bernardy et al., 2018), Idris 2 (Brady, 2021), as well as the experimental programming language Granule (Orchard et al., 2019).

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:

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

The first A is annotated with a grade 2, which in this context indicates that it *must* be used twice. Likewise, the types of y and z are graded with 0, enforcing zero usage, i.e., we are not allowed to use them in the body of f and must discard them. The result is that there is only one (normal form) inhabitant for this type: (x, x); the other assumptions will not even be considered in synthesis, allowing us to prune out branches which use resources in a way which violates the grades. In this example, these annotations take the form of natural numbers explaining how many times a value can be used, but we may instead wish to represent different kinds of program properties, such as sensitivity, strictness, or security levels for tracking non-interference, all of which are well-known instances of graded type systems (Orchard et al., 2019; Gaboardi et al., 2016a; Abel and Bernardy, 2020). Note that all of these examples are technically graded presentations of *coeffects*, tracking how a programs uses its context, in contrast with graded types for side *effects* (Orchard et al., 2014; Katsumata, 2014), which we do not consider here.

Hughes and Orchard (2020) developed a program synthesis technique for a linear type theory with graded modalities (of the form $\Box_r A$ where r is drawn from a semiring) and non-recursive types, building on proof search for linear logic (Hodas and Miller, 1994). We adapt some of their ideas to a setting which does not take a linear type theory as basis, but rather a type system in which grades are pervasive (such as the core of Haskell's linear types extension (Bernardy et al., 2018)) and which also contains user-defined 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 Hughes and Orchard to a fully graded setting (rather than linear setting) and also addressing recursion, recursive types, and

user-defined ADTs, none of which were previously considered in a linear or graded setting.
- We prove soundness, i.e., synthesised programs are typed by the goal type.
- We implement our synthesis calculus algorithmically as a tool for Granule's graded-base language extension.[1] For the most part, we elide the technicalities of the implementation but explain the connection with the theoretical development throughout.
- 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 (Osera and Zdancewic, 2015).
- Leveraging our calculus and implementation, we provide a prototype tool for synthesising Haskell programs written using GHC 9's linear types extension.

Road map Section 2 begins by providing a brief overview of the landscape of proof search in resourceful settings in, recalling the so-called 'resource management problem' which pervades all approaches to synthesis/proof search in this space.

We then introduce a core calculus with simple types and grades that forms the target language of our synthesis tool. This type system closely resembles other graded systems such as the coeffect calculus of Petricek et al. (2014), McBride's QTT (McBride, 2016; Atkey, 2018), the core of Linear Haskell (Bernardy et al., 2018), and the unified graded modal calculus of Abel and Bernardy (2020). We implemented the system defined here as a new language extension of the experimental programming language Granule (Orchard et al., 2019).

In Section 4, we present a calculus of synthesis rules for our language, showing how we ensure that constraints on resource usage are met and how we can use the information from grades to prune the search space of candidate programs. We also discuss some details of the implementation of our synthesis 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 (Andreoli, 1992).

We evaluate our implementation on a set of 46 benchmarks (Section 5), including several non-trivial programs which use algebraic data types and recursion. In Section 6, to demonstrate the practicality and versatility of our approach, we apply our algorithm to synthesising programs in Haskell from type signatures that use GHC's *linear types* extension, which is implemented underneath by a graded type system (Bernardy et al., 2018).

2 Overview of resourceful program synthesis

Section 1 discussed a rule for synthesising pairs and highlighted how graded types could be used to control the number of times assumptions are used in

[1] Available at: REDACTED

the synthesising term. In a linear or graded context, synthesis needs to handle the problem of *resource management* (Harland and Pym, 2000; Cervesato et al., 2000): how do we give a resourceful accounting to the context during synthesis so that we respect its constraints. Before explicating the full target language (Section 3) and our synthesis approach (Section 4), we give an overview of the main ideas here from the literature 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 (Girard, 1987):

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

Each subterm is typed by different contexts Γ_1 and Γ_2 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 inverting this typing rule into a synthesis rule yields:

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

As a declarative specification, the $\otimes_{\textsc{Intro}}$ synthesis rule is sufficient. However, this rule embeds a considerable amount of non-determinism when considered from an algorithmic perspective. Reading 'clockwise' starting from the bottom-left, given some context Γ and a goal $A \otimes B$, we have to split the context into disjoint subparts Γ_1 and Γ_2 such that $\Gamma = \Gamma_1, \Gamma_2$ in order to pass the Γ_1 and Γ_2 to the subgoals for A and B. For a context of size n there are 2^n possible such partitions! This quickly becomes intractable. Instead, Hodas and Miller (1994) developed a technique for linear logic programming, refined by Cervesato et al. (2000), where proof search for linear logic has both an *input context* of available resources and an *output context* of the remaining resources, which we write as judgements of the form $\Gamma \vdash A \Rightarrow^- t \mid \Gamma'$ for input context Γ and output context Γ'. Synthesis for multiplicative products then becomes:

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

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:

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

Thus this approach avoids the problem of having to split the input context, and facilitates efficient proof search for linear types. This idea was adapted by Hughes and Orchard (2020) to linear types augmented with graded modalities to

facilitate the synthesis of programs in Granule (Orchard et al., 2019). They called the above approach *subtractive resource management*, a style similar to *left-over* type-checking for linear type systems (Allais, 2018; Zalakain and Dardha, 2020). In a graded modal setting however this approach was shown to be costly.

Graded type systems, as we consider them here, have typing contexts in which free variables are assigned a type and a grade, usually drawn from some semiring parameterising the calculus. A useful example is the semiring of natural numbers for describing exactly how many times an assumption can be used (in contrast to linear assumptions which must be used exactly once). For example, the context $x :_2 A, y :_0 B$ explains that x must be used twice but y must never be used. The literature contains many examples of semirings for tracking other properties in this way, such as security labels (Gaboardi et al., 2016a; Abel and Bernardy, 2020), intervals of usage (Orchard et al., 2019), or hardware schedules (Ghica and Smith, 2014). In a graded setting, the subtractive approach is problematic as there is not necessarily a notion of subtraction for grades. Consider a version of the above example for subtractively synthesising a pair, but now for a context with some grades r and s on the input variables. Using a variable to synthesise a subterm now does not result in that variable being left out of the output context. Instead a new grade must be assigned in the output context that relates to the first by means of an additional constraint describing that some usage took place:

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

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

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

Hughes and Orchard implemented this approach, leveraging off-the-shelf SMT solving in the context of the Granule language, but show that a dual *additive* approach has much better performance. In the additive approach, output contexts describe what was *used* not what was is *left*. In the case of synthesising a term with multiple subterms (e.g. pairs), the output context from each premise is added together using the semiring addition operation applied pointwise on contexts to produce the final output in the conclusion. 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 into two prior to synthesising subterms. Hughes and Orchard evaluated both resource management strategies on a synthesis tool for 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 as part of synthesis; 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. Hughes and Orchard developed their approach for a graded linear calculus, where the base of the calculus is the linear λ-calculus with products, coproducts, and semiring-graded modalities on top. 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 goes further, in that it permits synthesis for arbitrary user-defined recursive ADTs in order to address more real-world synthesis problems.

3 Core calculus

We now define a core language with graded types. This system will be familiar to those who have encountered graded types before, drawing from the coeffect calculus of Petricek et al. (2014), Quantitative Type Theory (QTT) by McBride (2016) and refined by Atkey (2018) (though we omit dependent types from our language), the calculus of Abel and Bernardy (2020), and other graded dependent type theories (Moon et al., 2021). Similar systems also form the basis of the core of the linear types extension to Haskell (Bernardy et al., 2018). This calculus shares much in common with languages based on linear types, such as the graded monadic-comonadic calculus of (Gaboardi et al., 2016b), generalisations of Bounded Linear Logic (Brunel et al., 2014; Ghica and Smith, 2014), and Granule (Orchard et al., 2019) in the original 'linear base' form.

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

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

The function space $A^r \to B$ annotates the input type with a *grade* r drawn from a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ parameterising the calculus (where pre-ordering also requires that $+$ and $*$ are monotonic with respect to $\sqsubseteq$). Type constructors K include the unit, multiplicative linear product type, and linear coproduct type, which is also extended by names of user-defined ADTs in the implementation. The graded necessity modality $\Box_r A$ is similarly annotated by the grade r being an element of the semiring. Recursive types $\mu X.A$ are equi-recursive (although we also provide explicit typing rules) 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 (Milner, 1978). Subsequently, types may contain type variables α. Kinds κ are standard, given in the appendix.

The syntax of terms is given as:

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

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

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

Typing judgements have the form $\Sigma; \Gamma \vdash t : A$ assigning a type A to a term t under type variables Σ and variable context Γ, 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 constraints of the grade r. Structural exchange is permitted, allowing a context to be arbitrarily reordered. A global context D paramterises 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. We elide the (standard) details of kinding here.

Given a typing judgment $\Sigma; \Gamma \vdash t : A$ we say that t is both *well typed* and *well resourced* to highlight the role of grading in accounting for resource use via the semiring information. Another judgment types top-level terms (definitions) with polymorphic type schemes:

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

This rule takes the type scheme and adds its universally quantified type variables to Σ, where they can be used subsequently in the typing rules. The rule's premise then types the body at A, using the typing rules for terms of Figure 3.

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

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

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

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

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

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

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

Fig. 1: Typing rules for GrLang

Definition 1 (Scalar context multiplication).

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

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

Top-level definitions are typed by the Def rule. The definition x 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 $\Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')$ in a standard way as in Algorithm W (Milner, 1978).

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

Definition 2 (Context addition). *For all Γ_1, Γ_2* context addition *is defined as follows by ordered cases matching inductively on the structure of Γ_2:*

$$\Gamma_1 + \Gamma_2 = \begin{cases} \Gamma_1 & \Gamma_2 = \emptyset \\ ((\Gamma'_1, \Gamma''_1) + \Gamma'_2), x :_{(r+s)} A & \Gamma_2 = \Gamma'_2, x :_s A \wedge \Gamma_1 = \Gamma'_1, x :_r A, \Gamma''_1 \\ (\Gamma_1 + \Gamma'_2), x :_s A & \Gamma_2 = \Gamma'_2, x :_s A \wedge x \notin \mathsf{dom}(\Gamma_1) \end{cases}$$

For example, $(x :_1 A, y :_0 A) + x :_1 A = x :_{(1+1)} A, y :_0 A$. Thus, we can see $+$ on contexts as pointwise addition of the contexts via semiring $+$, taking the union of any disjoint parts. Returning to (App), we see that the context Γ_2 for

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

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

Fig. 2: Pattern typing rules of GrLang

the argument term is scaled by the grade of the function arrow r, as t_2 is used according to r the resulting term $t_1\ t_2$.

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

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

Introduction and elimination of data constructors is given by the Con and Case rules respectively, with Case also handling graded modality elimination via pattern matching. For Con, we may type a data constructor C of some data type $K\ \overline{A}$ (with zero or more type parameters represented by $\overline{A}$) if it is present in the global context of data constructors D. Data constructors are closed requiring our context Γ to have zero-use grades, thus we scale Γ by 0. Elimination of data constructors take place via pattern matching over a constructor. Patterns p are typed by the judgement $r \vdash\ p : A \rhd \Delta$ which states that a pattern p has type A and produces a context of typed binders Δ. The grade r to the left of the turnstile represents the grade information arising from usage in the context generated by this pattern match. The pattern typing rules are given by Figure 4.

Variable patterns are typed by PVar, which simply produces a singleton context containing an assumption $x :_r A$ from the variable pattern with any grade r. Pattern matching over data constructors is handled by the PCon rule. A data constructor may have up to zero or more sub-patterns ($p_1 ... p_n$), 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

[2] A discussion of this additional constraint on grades for case expressions is given by Hughes et al. (2020) comparing how this manifests in various approaches.

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 our promotion construct as a data constructor with the type $A^r \to \Box_r A$. We find it helpful to keep explicit modality introduction and elimination distinct from constructors, however, particularly with regard to synthesis.

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

Example 1. We have already seen examples of the natural numbers semiring with discrete ordering $(\mathbb{N}, *, 1, +, 0, \equiv)$, which counts exactly how many times variables are used. We denote this semiring as $\mathbb{N}_{\equiv}$ going forwards. This semiring is less useful in the presence of control-flow, e.g., when considering multiple branches in a case expression, where each branch uses a variable differently. A semiring of natural number intervals (Orchard et al., 2019) is more helpful here. An interval is given by a pair of natural numbers $\mathbb{N} \times \mathbb{N}$ written $r...s$. The lower bound of the interval is given by $r \in \mathbb{N}$ and the upper bound by $s \in \mathbb{N}$, with $0 = 0...0$ and $1 = 1...1$. Addition is defined pointwise and multiplication defined as in interval arithmetic, with ordering defined as $r...s \sqsubseteq r'...s' = r' \leq r \wedge s \leq s'$. This semiring allows us to write a function which performs an elimination on a coproduct (assuming $\mathsf{inl} : A^1 \to A \oplus B$, and $\mathsf{inr} : B^1 \to A \oplus B$ in D):

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

Example 2. The ! modality can be (almost) recovered via the {0,1,ω} semiring. For this semiring, we define $+$ as $r + s = r$ if $s = 0$, $r + s = s$ if $r = 0$, otherwise ω. Multiplication is $r \cdot 0 = 0 \cdot r = 0$, $r \cdot \omega = \omega \cdot r = \omega$ (where $r \neq 0$), and $r \cdot 1 = 1 \cdot r = r$. Ordering is defined as $0 \sqsubseteq \omega$ and $1 \sqsubseteq \omega$. This semiring allows us to express both linear and non-linear usage of values, with a 1 grade indicating linear use, 0 requires the value be discarded, and ω acting as linear logic's ! and permitting unrestrained use. This is similar to Linear Haskell's multiplicity annotations (although Linear Haskell has no equivalent of a 0 grade, only having `One` and `Many` annotations) (Bernardy et al., 2018). Using this semiring, we can write the k-combinator from the SKI calculus as:

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

Note however that some additional restrictions are required on pattern typing to get exactly the behaviour of ! with respect to products (Hughes et al., 2021). This is an orthogonal discussion and not relevant to the rest of this paper.

Lastly we note that the calculus enjoys admissibility of substitution (Abel and Bernardy, 2020) which is critical in type preservation proofs, and is needed in our proof of soundness for 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 judgements:

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

That is, given an input context Γ, for goal type A we can synthesise the term t with the output context Δ describing how variables were used in t. As with the typing rules, top-level definitions and data constructors in scope are contained in a set D, which paramterises the system. Σ is a context of kind-annotated type variables, which we elide in rules where it is simply passed inductively to the premise(s). The graded context Δ need not use all the variables in Γ, nor with exactly the same grades. Instead, the relationship between synthesis and typing is given by the central soundness result, which we state up-front:

Theorem 1 (Soundness of synthesis). *Given a particular pre-ordered semiring $\mathcal{R}$ parameterising the calculi, then:*

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

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

   i.e. t has type A under context Δ whose grades capture variable use in t.
2. *At the top-level, for all type schemes $\forall \overline{\alpha : \kappa}.A$ and terms t then:*

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

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 the grades in Γ. For example, a valid judgement (whose more general rule is seen shortly) under semiring $\mathbb{N}_{\equiv}$ is:

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

i.e., for goal A, if x has type A in the context then we synthesis x as the result program, regardless of the grades. A synthesis judgement such as this may be part of a larger derivation in which the grades eventually match, i.e., this judgement forms part of a larger derivation which has a further subderivation in which x is used again and thus the total usage for x is eventually 2 as prescribed by the input context. However, at the level of an individual judgement

we do not guarantee that the synthesised term is well-resourced. A reasonable *pruning condition* that could be used to assess whether any synthesis judgement is *potentially* well-resourced is $\exists \Delta'.(\Delta + \Delta') \sqsubseteq \Gamma$, i.e., there is some additional usage Δ' (that might come from further on in the synthesis process) that 'fills the gap' in resource use to produce $\Delta + \Delta'$ which is overapproximated by Γ. In this example, $\Delta' = x :_1 A$ would satisfy this constraint, explaining that there is some further possible single usage which will satisfy the incoming grade. However, previous work on graded linear types showed that excessive pruning at every step becomes too costly in a general setting (Hughes and Orchard, 2020). 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).

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

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 judgements, where meta-level terms to the left of $\Rightarrow$ are inputs (i.e., context Γ and goal type A) and terms to the right of $\Rightarrow$ are outputs (i.e., the synthesised term t and the usage context Δ). Whilst we largely present the approach here in abstract terms, via the synthesis judgements, 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 the TopLevel rule, which is for a judgment form with a type scheme goal instead of just a type, providing the entry-point to synthesis:

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

This rule takes the universally quantified type variables $\overline{\alpha : \kappa}$ from the type scheme and adds them to the type variable context Σ; type variables are only equal to themselves. This rule corresponds to the generalisation step of typing polymorphic definitions (Milner, 1978).

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

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

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 the synthesis of terms which may not be well-resourced, e.g., if $r = 0$, the rule still synthesises a use of x. This is locally ill-resourced, but is acceptable at the global level as we check that an assumption has been used correctly in the rule where the assumption is bound. This does leads us to consider some branches of synthesis that are guaranteed to fail: at the point of synthesising a usage of a variable in the additive scheme, isolated from information about how else the variable is used, there is no way of knowing if such a usage will be permissible in the final synthesised program. However, it also reduces the amount of intermediate theorems that need solving, which can significantly effect performance as shown by Hughes and Orchard (2020), especially since the variable rule is applied very frequently.

Functions Synthesis of programs from function types is handled by the $\to_{\mathrm{R}}$ and $\to_{\mathrm{L}}$ rules, which synthesise abstraction and application terms, respectively. An abstraction is synthesised like so:

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

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 *approximates* q. If met, Δ becomes the output context of the rule's conclusion.

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

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

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

1. The first premise synthesises a term t_1 from the goal type C under the assumption that the function x_1 has been applied and its result is bound to x_2. This placeholder assumption is bound with the same grade as x_1.
2. The second premise synthesises an argument t_2 of type A for the function x_1. In the implementation, this synthesis step occurs only after a term t_1 is found for the goal C as a heuristic to avoid possibly unnecessary work if no term can be synthesised for C anyway.

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

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

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

Using polymorphic definitions Programs can be synthesised from a polymorphic type scheme (the previously shown TopLevel rule), treating universally-quantified type varibles at the top-level of our goal type as logical atoms which cannot be unified with and are only equal to themselves. The Def rule handles the synthesis of a *use* of a top-level polymorphic function via instantiation:

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

For example, in the following we have a polymorphic function `flip` that we want to use to synthesise a monomorphic function:

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

To synthesise the term `flip x`, the type scheme of `flip` is instantiated via Def with $\emptyset \vdash (Int \otimes Int)^1 \to (Int \otimes Int) = \mathrm{inst}(\forall c : \mathrm{Type}, d : \mathrm{Type}.(c \otimes d)^1 \to (d \otimes c))$.

Graded modalities Graded modalities are introduced and eliminated explicitly through the $\Box_{\mathrm{R}}$ and $\Box_{\mathrm{L}}$ rules, respectively. In the $\Box_{\mathrm{R}}$ rule, we synthesise a

promotion $[t]$ for some graded modal goal type $\Box_r A$:

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

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

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 the assumption x of graded modal type $\Box_q A$, we bind a fresh assumption in the synthesis of 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 $(,) : A^1 \to B^1 \to A \otimes B$, $Unit : \mathsf{Unit}$, $\mathsf{inl} : A^1 \to A \oplus B$, 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.

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

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

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

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

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

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

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

This leaves the question of how we calculate the final grade to attribute to x in the output context of the rule's conclusion. For each bound assumption, we generate a fresh grade variable ${s'}_j^i$ 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 $\exists {s'}_j^i.\ s_j^i \sqsubseteq {s'}_j^i \cdot q_j^i \sqsubseteq r \cdot q_j^i$. The join of each ${s'}_j^i$ (for each assumption) is then taken to form a grade variable s_i which represents the total

usage of x for this branch that arises from the use of assumptions which were bound via the pattern match (i.e. not usage that arises from reusing x explicitly inside t_i). For the output context of the conclusion, we then take the join of output context from the constructors used. This is extended with the original x assumption with the output grade consisting of the join of each r_i (the usages of x directly in each branch) plus the join of each s_i (the usages of the assumptions that were bound from matching on a constructor of x).

*Example 3 (Example of **case** synthesis).* Consider two possible synthesis results:

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

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

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

$$\frac{\begin{array}{c}\mathsf{Just} : \forall \overline{\alpha : \kappa}.A'^{1} \to A' \oplus \mathsf{Unit} \in D \\ \mathsf{Nothing} : \forall \overline{\alpha : \kappa}.\mathsf{Unit}^{1} \to A' \oplus \mathsf{Unit} \in D \\ \Sigma \vdash A^{1} \to A \oplus \mathsf{Unit} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.A'^{1} \to A' \oplus \mathsf{Unit}) \\ \Sigma \vdash \mathsf{Unit}^{1} \to A \oplus \mathsf{Unit} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.\mathsf{Unit}^{1} \to A' \oplus \mathsf{Unit}) \\ (1) \quad \Sigma; x :_r \mathsf{Unit} \oplus A, y :_s A, z :_{r \cdot q_1} A \vdash A \Rightarrow z \mid x :_0 \mathsf{Unit} \oplus A, y :_0 A, z :_{s_1} A \\ (2) \quad \Sigma; x :_r \mathsf{Unit} \oplus A, y :_s A \vdash A \Rightarrow y \mid x :_0 \mathsf{Unit} \oplus A, y :_1 A \\ \exists s'_1.\, s_1 \sqsubseteq s'_1 \cdot q_1 \sqsubseteq r \cdot q_1 \qquad s' = s'_1\end{array}}{\Sigma; x :_r \mathsf{Unit} \oplus A, y :_s A \vdash A \Rightarrow (\mathbf{case}\ x\ \mathbf{of}\ \mathsf{Just}\ z \to z; \mathsf{Nothing} \to y) \mid x :_{(0 \sqcup 0) + s'} \mathsf{Unit} \oplus A, y :_{0 \sqcup 1} A}\ \textsc{Case}$$

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

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

These constraints can be satisfied with the natural-number intervals semiring where y has grade 0..1 and x has grade 1..1.

Deep pattern matching, over nested data constructors, is handled via inductively applying the $\mathrm{C_L}$ rule but with a post-synthesis refactoring procedure substituting the pattern match of the inner case statement into the outer pattern match. For example, nested matching on pairs becomes a single **case** with nested pattern matching, simplifying the program:

$$\begin{array}{rl} & \mathbf{case}\ x\ \mathbf{of}\ (y_1, y_2) \to \mathbf{case}\ y_1\ \mathbf{of}\ (z_1, z_2) \to z_2 \\ \textit{(rewritten to)} \rightsquigarrow & \mathbf{case}\ x\ \mathbf{of}\ ((z_1, z_2), y_2) \to z_2 \end{array}$$

Recursion Synthesis permits recursive definitions, as well as programs which may make use of calls to functions from a user-supplied context of function definitions in scope (see Section 4.2). Synthesis of non-recursive function applications may take place arbitrarily, however, synthesising a recursive function definition application requires more care. To ensure that a synthesised programs terminates, we only permit synthesis of terms which are *structurally recursive*,

i.e., those which apply the recursive definition to a subterm of the function's inputs (Osera, 2015).

Synthesis rules for recursive types (μ-types) are fairly straightforward:[3]

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

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

Recursive data structures present a challenge in the implementation. For our list data type, how do we prevent our synthesis tool from simply applying the μ_{L} rule, followed by the C_{L} rule on the `Cons` constructor ad infinitum? We resolve this issue using an *iterative deepening* approach to synthesis similar to the approach used by Myth (Osera, 2015). Programs are synthesised with elimination (and introduction) forms of constructors restricted up to a given depth. If no program is synthesised within these bounds, then the depth limits are incremented. Combined with focusing (see Section 4.3), this provides the basis for an efficient implementation of the above rules.

4.2 Input-output examples

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

We augmented the Granule language with first-class syntax for specifying input-output examples, both as a feature for aiding synthesis but also for aiding documentation that is type checked (and therefore more likely to stay consistent

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

with a code base as it evolves). Synthesis specifications are written in Granule directly above a program hole (written using `?`) using the `spec` keyword. The input-output examples are then listed per-line.

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

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

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

with the following resulting program produced by our synthesis algorithm (on average in about 400ms on a standard laptop, see Section 5 where this is one of the benchmarks for evaluation):

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

4.3 Focusing proof search

The calculus presented above serves as a starting point for implementing a synthesis algorithm in Granule. However, the rules are highly non-deterministic with regards the order in which they may be applied. For example, after applying a $(\rightarrow_{\mathrm{R}})$-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* (Andreoli, 1992). Focusing is a tool from linear logic proof theory based on the idea that some rules are invertible, i.e., whenever the conclusion of the rule is derivable, then so are the premises. In other words, the order in which we apply invertible rules doesn't matter. By fixing a particular ordering on the application of invertible rules, we eliminate much of the non-determinism that arises from trying branches which differ only in the order in which invertible rules are applied. We apply focusing to our calculus, which

forms the basis of our implementation. The full focusing versions of the rules from our calculus, and their proof of soundness, can be found in the appendix.

Our implementation also relies on the use of backtracking proof search, leveraging a monadic interface (Kiselyov et al., 2005). A resulting synthesised program can be rejected and synthesis required to produce an alternate result (what we call a *retry*) via backtracking.

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 data structures such as lists, streams, booleans, option ('maybe') types, unary natural numbers, and binary trees. We draw many of these from the benchmark suite of the Myth synthesis tool (Osera and Zdancewic, 2015). Benchmarks are divided into classes based on the main data type, with an additional category of miscellaneous programs. The type schemes for the full suite of benchmarks can be found in Appendix C.

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 goal 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 typed-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 provides a means

to check whether a program may be 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-well-resourced outputs) 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 non-graded Cartesian setting. Comparing our tool for speed against another type-directed (but not graded-directed) synthesis tool is likely to be largely uninformative due to differences in implementation approach obscuring any meaningful comparison. Thus, we instead compare timings for the graded approach and de-graded approach within Granule. We also record the number of search paths taken (over all retries) to assess the level of pruning in the graded vs non-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 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 a program 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) N needed to reach a well-resourced program. In all cases, this resulting program in the Cartesian column was equivalent to

that generated by the graded synthesis, none of which needed any retries (i.e., implicitly $N = 0$ for graded synthesis). 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. The results show that 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) it still outperforms the graded equivalent. This appears to be due to the cost of our SMT solving procedure which must first compile a first-order theorem on grades into the SMT-lib file format, start up Z3, and then run the solver. Considerable amounts of system overhead are incurred in this procedure. A more efficient implementation calling Z3 directly (e.g., 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, as well as `sum` for both lists and trees. In each of these cases, 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).

Interestingly the paths explored are sometimes the same because we use backtracking search in the Cartesian+Graded type check case where, if an output program fails to type check against the graded type signature, the search backtracks rather than starting again.

Confirming H2, we find that for the non-graded setting without graded type-checking, further examples are required 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 was required.

				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

We briefly examine some of the more complex benchmarks which make use of almost all of our synthesis rules in one program. The `stutter` case from the List class of benchmarks is specified as:

```
stutter : ∀ a . List (a [2]) %1..∞ → List a
spec
    stutter % 0..∞
stutter = ?
```

This is a function which takes a list of values of type `a`, where each element in the list is explicitly graded by `2`, indicating that each element must be used twice. The return type of `stutter` is a list of type `a`. The argument list itself must be used at least once with potential usage extending up to infinity, suggesting that some recursion will be necessary in the program. This is further emphasised by the `spec`, which states that we can use the definition of `stutter` inside the function body in an unrestricted way. From this, we synthesise the program:

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

in 1325ms (∼1.3 seconds). We also have a `stutter` case in the Tree class of benchmarks, which instead performs the above transformation over a binary tree data type, which yields the following program in 693ms (∼0.7 seconds):

```
stutter : ∀ a b . Tree (a [2]) % 1..∞ → Tree (a, a)
spec
    stutter %0..∞
stutter Leaf = Leaf;
stutter (Node y [v] u) = ((Node (stutter y)) (v, v)) (stutter u)
```

Lastly, we compare between the number of examples required by Granule (using grades) and the Myth program synthesis tool (based on pruning by examples). We take the cases from our benchmark set which have an equivalent in the Myth benchmark suite (Osera and Zdancewic, 2015). Table 2 shows the number of input-output examples used in Granule, and the number required for the equivalent Myth case. In both Granule and Myth this number represents the minimal number of examples required to synthesise the correct program.

		Granule	Myth
Problem		#/Exs	#/Exs
List	append	0	6
	concat	1	6
	snoc	1	8
	drop	1	13
	inc	1	4
	head	1	3
	tail	1	3
	last	1	6
	length	1	3
	map	0	8
	stutter	0	3
	sum	1	3

		Granule	Myth
Problem		#/Exs	#/Exs
Bool	neg	2	2
	and	4	4
	impl	4	4
	or	4	4
	xor	4	4
Nat	isEven	2	4
	pred	1	3
Tree	map	0	7

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

For most of the problems (15 out of 20), Granule required fewer examples to identify the desired program in synthesis. The disparity in the number of examples required is quite significant in some cases: with 13 examples required by Myth to synthesise the *concat* problem but only 1 example for Granule. This shows the pruning power of graded information in synthesis, confirming H2.

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 of GHC's existing type system (Bernardy et al., 2018) (the `LinearType` extension). This system is closely related to Granule, but limited only to one semiring for its grades. This however presents an exciting opportunity: can we leverage our tool to synthesise (linear) Haskell programs?

Like Granule, grades in Linear 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 for (`'Many`) unrestricted usage. Likewise, in Granule, we can model linear types using the 0-1-ω semiring (see example 2) (Hughes et al., 2021).

Synthesising Linear Haskell programs then simply becomes a task of parsing a Haskell type into a Granule equivalent, synthesising a term from this, and compiling the synthesised term back into Haskell. The close syntactic correspondence between Granule and Haskell makes this translation straightforward.

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.

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

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

```
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

Subtractive resource management Prior to introducing the additive resource management scheme in Section 2, we first considered a 'subtractive approach'. Ultimately, we opted not to use this scheme as the basis for our synthesis calculus due to concerns regarding the efficiency of its implementation.

In the subtractive scheme, the VAR rule generates a constraint which determines if the use of a variable is permissible based on the rest of the partially synthesised program:

$$\frac{\exists s.\ r \sqsupseteq s + 1}{\Gamma, x :_r A \vdash A \Rightarrow x \mid \Gamma, x :_s A}\ \textsc{Var}$$

i.e., r must overapproximate one use of x plus the future use of x given by the existential s.

A comparative evaluation of the additive and subtractive schemes by Hughes and Orchard (2020) showed that these, and other associated constraints from the subtractive approach, are larger, typically more complex, and are discharged more frequently than their counterparts in the additive system (every time a variable usage is being considered in the above case). Their evaluation concluded that the only situation where subtractive decisively outperformed additive was on purely linear programs. This, coupled with the fact that the subtractive approach has limitations in the presence of polymorphic grades, influenced our decision to adopt the additive scheme, especially as we considered much more complex programs than Hughes and Orchard (2020), e.g., targeting recursion.

Related work Section 2 considered some of the related work. Hughes and Orchard (2020) target a 'linear base' calculus: the linearly-typed λ-calculus (with linear function types $\multimap$) and explicit introduction and elimination of graded modalities. They introduced the notion of additive resource management, whose philosophy we adopted here, but in our setting of a fully-graded ('graded base') calculus which makes the majority of the rules considerably different. In particular, the synthesis rule for elimination data types is much more complex here: in a linear setting many of the constraints and grades specialise away as essentially being equal to 1 (and Hughes and Orchard considered only linear products and coproducts with their own specialised eliminators).

There are terms we can synthesise in the graded-base work of this paper which would not be possible in Hughes and Orchard (even if we restricted our target language to match their simplified typing calculus) due to incresed expressivity here. For example, synthesis of programs which perform "deep" pattern matching over a graded data structure is not possible in their system:

```
1 deep : ∀ a b : Type . (a, (a, b)) [0..1] ⊸ b
2 deep = ?
```

The linear-base setting of Hughes and Orchard cannot synthesise a solution here because it cannot nest pattern matches to inherit the grade `0..1` on the components of the pair `a` and `b`. However, in our system the pattern matching synthesis rules mean we can synthesise (in just a few steps):

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

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

In other recent work, Fiala et al. (2023) synthesise Rust programs from types. Rust's type system is augmented by the borrow checker which provides notions of ownership and borrowing (Jung et al., 2019). Fiala et al. (2023) develop a custom program logic *Synthetic Ownership Logic* (a variant of a separation logic) that integrates a typed approach to Rust ownership (as well as functional specifications) allowing synthesis to follow a deductive approach. There is some philosophical overlap with the resourceful ideas inherent 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 et al. (2022) show that uniqueness types (Smetsers et al., 1994) can be implemented as an extension of a linear type theory with a non-linearity modality, introducing an additional uniqueness modality. Our work could be straightforwardly extended to this setting to provide synthesis for uniqueness types as a precursor to the full ownership and borrowing system of Rust. Studying this, and the comparing to the recent approach of (Fiala et al., 2023) is future work.

The Idris dependently typed language provides automated proof search as part of its implementation (Brady, 2013). In Idris 2, the core type theory is based on Quantitative Type Theory (Atkey, 2018; McBride, 2016), with the 0-1-ω semiring (Example 2) and with proof synthesis extended to utilise these grades (Brady, 2021). This approach has some relation to ours, but in a more limited single-semiring setting. Furthemore, it cannot synthesise entire recursive functions. However our approach is readily applicable in this situation; targeting Idris is future work.

Conclusion A logical next step is to incorporate GADTs (Generalised ADTs), i.e., indexed types, into the synthesis algorithm. Granule provides support for user-defined GADTs, and the interaction between grades and type indices is a key contributor to Granule's expressive power (Orchard et al., 2019). Consider our list type benchmarks for example. In most cases, when we want to synthesise a recursive function definition which takes a list as input, we have to give the list a $0..\infty$ interval grade to account for potentially unlimited usage. Take for example the `map` benchmark:

```
map : ∀ a b . (a % 1..∞ → b) % 0..∞ → List a % 1..∞ → List b
spec
    map % 0..∞
map = ?
```

With indexed types, we can be much more precise. Consider a size-indexed vector type, defined as:

```
data Vec (n : Nat) t where
  Nil  : Vec 0 t;
  Cons : t → Vec n t → Vec (n+1) t
```

The corresponding `map` function can be given a much tighter specification, connecting the usage of the input function to the length of the vector, from which we could synthesise the program:

```
vmap : ∀ {a b : Type, n : Nat} . (a → b) %n → Vec n a → Vec n b
spec
    vmap % n
vmap f Nil = Nil;
vmap f (Cons x xs) = Cons (f x) (vmap f xs)
```

The latter type not only provides us with a greater opportunity to prune grade-violating programs, its type is also much more descriptive of the user's intent. Adapting our approach to GADTs is future work, and mostly consists in extending the typing for **case** to handle GADT specialisation.

Our goal is to create a program synthesis tool for Granule which assists the programmer in writing programs with resource-sensitive types. We intend to pursue further improvements to our tool which serve this end, including reducing the overhead of SMT solving, integrating examples into the search algorithm itself in the style of Myth (Osera and Zdancewic, 2015) and Leon (Kneuss et al., 2013), as well as considering possible semiring-dependent optimisations that may be applicable.

With the rise in LLMs (Large Language Models) showing their power at program synthesis tasks (Austin et al., 2021; Jain et al., 2021), the deductive approach still has something to contribute: it provides correct-by-construction synthesis based on specifications, rather than predicted programs which may violate more fine-grained type constraints (e.g., as provided by grades). Future approaches may combine both LLM approaches with deductive approaches, where the logical engine of the deductive approach can guide prediction. Exploring this is further work and a general opportunity and challenge for the deductive synthesis community.

Bibliography

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, URL https://doi.org/10.1145/3408972

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, URL https://doi.org/10.1007/978-3-642-39799-8_67

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)

Andreoli, J.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation **2**(3), 297–347 (06 1992), ISSN 0955-792X, https://doi.org/10.1093/logcom/2.3.297

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, URL https://doi.org/10.1145/3209108.3209189

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), URL https://arxiv.org/abs/2108.07732

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, URL https://doi.org/10.1145/3158093

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, URL https://doi.org/10.1017/S095679681300018X

Brady, E.C.: Idris 2: Quantitative type theory in practice. CoRR **abs/2104.00480** (2021), URL https://arxiv.org/abs/2104.00480

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

Cervesato, I., Hodas, J.S., Pfenning, F.: Efficient resource management for linear logic proof search. Theoretical Computer Science **232**(1), 133 – 163 (2000), ISSN 0304-3975, https://doi.org/https://doi.org/10.1016/S0304-3975(99)00173-5

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, URL https://doi.org/10.1145/3434331

Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. SIGPLAN Not. **50**(6), 229–239 (jun 2015), ISSN 0362-1340, https://doi.org/10.1145/2813885.2737977, URL https://doi.org/10.1145/2813885.2737977

Fiala, J., Itzhaky, S., Müller, P., Polikarpova, N., Sergey, I.: Leveraging Rust Types for Program Synthesis. To appear in the Proceedings of PLDI (2023)

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

Gaboardi, M., Katsumata, S., Orchard, D.A., Breuvart, F., Uustalu, T.: Combining effects and coeffects via grading. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016, pp. 476–489, ACM (2016a), https://doi.org/10.1145/2951913.2951939

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 (2016b), ISBN 9781450342193, https://doi.org/10.1145/2951913.2951939, URL https://doi.org/10.1145/2951913.2951939

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

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

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)

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)

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

Hodas, J., Miller, D.: Logic Programming in a Fragment of Intuitionistic Linear Logic. Information and Computation **110**(2), 327 – 365 (1994), ISSN 0890-5401, https://doi.org/https://doi.org/10.1006/inco.1994.1036

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), URL https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271465

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, URL https://doi.org/10.1007/978-3-030-68446-4_8

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, URL https://doi.org/10.4204/EPTCS.353.6

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

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)

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

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), ISSN 0362-1340, https://doi.org/10.1145/1090189.1086390

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), ISBN 9781450323741, https://doi.org/10.1145/2509136.2509555, URL https://doi.org/10.1145/2509136.2509555

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

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

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, URL https://doi.org/10.1007/978-3-030-99336-8_13

McBride, C.: I Got Plenty o' Nuttin', pp. 207–233. Springer International Publishing, Cham (2016), ISBN 978-3-319-30936-1, https://doi.org/10.1007/978-3-319-30936-1‟˙12

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

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, URL https://doi.org/10.1007/978-3-030-72019-3_17

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

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

Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. SIGPLAN Not. **50**(6), 619–630 (Jun 2015), ISSN 0362-1340, https://doi.org/10.1145/2813885.2738007

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

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

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), ISBN 9781450342612, https://doi.org/10.1145/2908080.2908093, URL https://doi.org/10.1145/2908080.2908093

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), ISBN 978-3-540-48333-5, https://doi.org/10.1007/3-540-57787-4‟˙23

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

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