Resourceful Program Synthesis from Graded Linear Types

Jack Hughes[0000−0003−3174−4689] and Dominic Orchard[0000−0002−7058−7842]

School of Computing, University of Kent

Abstract. Linear types provide a way to constrain programs by specifying that some values must be used exactly once. Recent work on *graded modal types* augments and refines this notion, enabling fine-grained, quantitative specification of data use in programs. The information provided by graded modal types appears to be useful for type-directed program synthesis, where these additional constraints can be used to prune the search space of candidate programs. We explore one of the major implementation challenges of a synthesis algorithm in this setting: how does the synthesis algorithm efficiently ensure that resource constraints are satisfied throughout program generation? We provide two solutions to this *resource management* problem, adapting Hodas and Miller's input-output model of linear context management to a graded modal linear type theory. We evaluate the performance of both approaches via their implementation as a program synthesis tool for the programming language Granule, which provides linear and graded modal typing.

1 Introduction

Type-directed program synthesis is a long-studied technique rooted in automated theorem proving [29]. A type-directed synthesis algorithm can be constructed as an inversion of type checking, starting from a type and inductively synthesising well-typed subterms, pruning the search space via typing. Via the Curry-Howard correspondence [21], we can view this as proof search in a corresponding logic, where the goal type is a proposition and the synthesised program is its proof. Recent work has extended type-directed synthesis to refinement types [34], cost specifications [27], differential privacy [35], and example-guided synthesis [12,33].

Automated proof search techniques have been previously adapted to linear logics, accounting for resource-sensitive reasoning [7–9,20,31]. By removing the structural rules of contraction and weakening, linear logic allows propositions to be treated as resources that must be used exactly once [17]. Non-linear propositions are captured via the 'exponential' modality !. Linearity introduces a new dimension to proof search and program synthesis: how do we inductively generate terms whilst pruning the search space of those which violate linearity? For example, consider the following inductive *synthesis rule*, mirroring Gentzen's sequent calculus [15], which synthesises a term of type $A \otimes B$:

$$\frac{\Gamma_1 \vdash A \Rightarrow t_1 \qquad \Gamma_2 \vdash B \Rightarrow t_2}{\Gamma_1, \Gamma_2 \vdash A \otimes B \Rightarrow \langle t_1, t_2 \rangle}\ \textsc{Pair}$$

Reading the rule *bottom up*: from a context of assumptions Γ_1, Γ_2 we can synthesise the pair $\langle t_1, t_2 \rangle$ from the product type $A \otimes B$ provided that we can inductively synthesise the subterms of the pair, using Γ_1 for the left side and Γ_2 for the right.

But how do we partition a context of free variables Γ into Γ_1 and Γ_2 such that Γ_1 contains only those variables needed by t_1 and Γ_2 only those for t_2? A naïve approach is to try every possible partition of Γ. However, this becomes unmanageable as the number of possible partitions is $2^{|\Gamma|}$, i.e., exponential in the number of assumptions. This issue has been explored in automated theorem proving for linear logic, and is termed the *resource management problem* [7].

To address this, Hodas and Miller described an *input-output context management* scheme for linear logic programming [20], further developed by Cervesato et al. [7]. In this approach, synthesis rules take the form $\Gamma \vdash A \Rightarrow t;\ \Delta$ with an *input context* Γ and an *output context* Δ which contains all the hypotheses of Γ that were not used in the proof t of A (akin to the notion of *left over* typing for linear type systems [2, 36]). This output context is then used as the input context to subsequent subgoals. In the case of $A \otimes B$, synthesis has the form:

$$\frac{\Gamma \vdash A \Rightarrow t_1;\ \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow t_2;\ \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow \langle t_1, t_2 \rangle;\ \Delta_2}\ \textsc{Pair\_LeftOver}$$

The non-determinism of how to divide Γ is resolved by using the entire context as the input for the synthesis of the first subterm t_1 from type A. If this succeeds, the context Δ_1 is returned containing the resources not needed to construct t_1. These remaining resources provide the input context to synthesise t_2 from B, which in turn returns an output context Δ_2 containing the resources not used by the pair $\langle t_1, t_2 \rangle$. We extend this approach, which we term *subtractive resource management*, to *graded modal types* and present its dual: *additive resource management*. In the additive approach, the output context describes what resources were used to synthesise a term, rather than what may still be used.

Graded modal types comprise an indexed family of modal operators whose indices have structure capturing program properties [32]. In the context of linear logic, graded modalities generalise the indexed modality of Bounded Linear Logic [18] $!_r A$ where $r \in \mathbb{N}$ captures the upper bound r on the number of times A is used. Generalising such indices to an arbitrary (pre-ordered) semiring yields a type system which can be instantiated to track various properties via the graded modality, a technique which is increasingly popular [4, 13, 14, 16, 24, 25, 32, 36].

Our primary contribution is the extension of the input-output model of resource management for linear program synthesis to graded modal types. Our input and output contexts contain both linear and graded assumptions. Graded assumptions are annotated with a *grade*: an element of a pre-ordered semiring describing the variable's use. For example, grades drawn from $\mathbb{N}$ yield a system akin to BLL which counts the number of times a variable is used, where a graded assumption $x : [A]_2$ means x can be used twice. An example instantiation of our subtractive pair introduction rule is then as follows:

$$\frac{\Gamma, x : [A]_2 \vdash A \Rightarrow x;\ \Gamma, x : [A]_1 \qquad \Gamma, x : [A]_1 \vdash A \Rightarrow x;\ \Gamma, x : [A]_0}{\Gamma, x : [A]_2 \vdash A \otimes A \Rightarrow \langle x, x \rangle;\ \Gamma, x : [A]_0}$$

The initial input context contains graded assumption $x : [A]_2$. The first premise synthesises the term x, returning an output context which contains the assumption x with grade 1, indicating that x has been used once and can be used one more time. The next premise synthesises the second part of the pair as x using its remaining use. In the final output context, x is graded by 0, preventing it from being used to synthesise subsequent terms.

We adapt the input-output model of linear logic synthesis to subtractive and additive approaches in the presence of graded modal types, pruning the search space via the quantitative constraints of grades. We develop a type-directed synthesis tool for Granule, a functional language which combines indexed, linear, and graded modal types [32]. Granule supports various graded modalities, and its type checker leverages the Z3 SMT solver to discharge constraints on grades [30]. As type-based synthesis follows the structure of types, it is necessary to solve equations on grades during synthesis, for which we make use of Granule's SMT integration. Such calls to an external prover are costly, and thus efficiency of resource management is a key concern.

Section 2 introduces our core type theory (a subset of Granule's type system) based on the linear λ-calculus extended with graded modal types, pairs, and sums. Section 3 describes the two core synthesis calculi (subtractive and additive) as augmented inversions of the typing rules, as well as a variant of additive synthesis. Section 4 describes the implementation[1] and gives a quantitative comparison of the synthesis techniques on a suite of benchmark programs. The main finding is that the additive approach is often more efficient than the subtractive, presenting a departure from the literature on linear logic theorem proving which is typically subtractive.

Throughout, we will tend towards using *types-and-programs* terminology rather than *propositions-and-proofs*. Through the Curry-Howard correspondence, one can switch smoothly to viewing our approach as proof search in logic.

2 Graded Linear λ-calculus

Our focus is a linear λ-calculus akin to a simply-typed linear functional language with graded modalities, resembling the core languages of Gaboardi et al. [14] and Brunel et al. [4], and a simply-typed subset of Granule [32].

Types comprise linear functions, multiplicative conjunction (product types $\otimes$ and unit 1), additive disjunction (sum types $\oplus$), and a *graded modality* $\Box_r$:

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

where $\Box_r A$ is an indexed family of type operators where r ranges over the elements of some pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ parameterising the calculus (where $*$ and $+$ are monotonic with respect to the pre-order $\sqsubseteq$).

[1] `https://github.com/granule-project/granule/releases/tag/v0.8.0.0`

The syntax of terms provides the elimination and introduction forms:

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

We use the syntax $()$ for the inhabitant of multiplicative unit 1. Pattern matching via a **let** is used to eliminate products and unit types; for sum types, **case** is used to distinguish the constructors. The construct $[t]$ introduces a graded modal type $\Box_r A$ by 'promoting' a term t to the graded modality, and $\textbf{let}\, [x] = t_1\, \textbf{in}\, t_2$ eliminates a graded modal value t_1, binding a graded variable x in scope of t_2.

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

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

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

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

Definition 1 (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:*

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

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

Definition 2 (Partial least-upper bounds of contexts). *For all Γ_1, Γ_2:*

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

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

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

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

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

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \quad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{Let}\Box \qquad \frac{}{\emptyset \vdash () : 1}\ 1 \qquad \frac{\Gamma_1 \vdash t_1 : 1 \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : A}\ \textsc{Let1}$$

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

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \quad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{Approx} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathbf{inl}\,t : A \oplus B}\ \textsc{Inl} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathbf{inr}\,t : A \oplus B}\ \textsc{Inr}$$

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

Fig. 1. Typing rules of the graded linear λ-calculus

Figure 1 defines the typing rules. Linear variables are typed in a singleton context (Var). Abstraction (Abs) and application (App) follow the rules of the linear λ-calculus. Rules for multiplicative products (pairs) and additive co-products (sums) are routine, where pair introduction (Pair) adds the contexts used to type the pair's constituent subterms. Pair elimination (LetPair) binds a pair's components to two linear variables in the scope of the body t_2. The Inl and Inr rules handle the typing of constructors for the sum type $A \oplus B$. Elimination of sums (Case) takes the least upper bound (defined above) of the contexts used to type the two branches of the case.

The Weak rule captures weakening of assumptions graded by 0 (where $[\Delta]_0$ denotes a context containing only graded assumptions graded by 0). Dereliction (Der), allows a linear assumption to be converted to a graded assumption with grade 1. Grade approximation is captured by the Approx rule, which allows a grade r to be converted to another grade s, providing that r is *approximated by* s, where the relation $\sqsubseteq$ is the pre-order provided with the semiring. Introduction and elimination of the graded modality is provided by the Pr and Let rules respectively. The Pr rule propagates the grade r to the assumptions through *scalar multiplication* of Γ by r where every assumption in Γ must already be graded (written $[\Gamma]$ in the rule), defined:

Definition 3 (Scalar context multiplication).

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

The Let rule eliminates a graded modal value $\Box_r A$ into a graded assumption $x : [A]_r$ with a matching grade in the scope of the **let** body.

We now give three examples of different graded modalities.

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

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

The graded modal value z' captures the 'capability' for a value of type A to be used twice. This capability is made available by eliminating $\Box$ (via **let**) to the variable z, which is graded $z : [A]_2$ in the scope of the body.

Example 2. Exact usage analysis is less useful when control-flow is involved, e.g., eliminating sum types where each control-flow branch uses variables differently. The above $\mathbb{N}$-semiring can be imbued with a notion of *approximation* via less-than-equal ordering, providing upper bounds. A more expressive semiring is that of natural number intervals [32], given by pairs $\mathbb{N} \times \mathbb{N}$ written $[r...s]$ here for the lower-bound $r \in \mathbb{N}$ and upper-bound usage $s \in \mathbb{N}$ with $0 = [0...0]$ and $1 = [1...1]$, addition and multiplication defined pointwise, and ordering $[r...s] \sqsubseteq [r'...s'] = r' \leq r \wedge s \leq s'$. Then a coproduct elimination function can be written and typed:

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

Linear logic's exponential $!A$ is given by $\Box_{[0...\infty]}A$ with intervals over $\mathbb{N} \cup \{\infty\}$ where ∞ is absorbing for all operations, except multiplying by 0.

Example 3. Graded modalities can capture a form of information-flow security, tracking the flow of labelled data through a program [32], with a lattice-based semiring on $\mathcal{R} = \{\mathsf{Unused} \sqsubseteq \mathsf{Hi} \sqsubseteq \mathsf{Lo}\}$ where $0 = \mathsf{Unused}$, $1 = \mathsf{Hi}$, $+ = \sqcup$ and if $r = \mathsf{Unused}$ or $s = \mathsf{Unused}$ then $r * s = \mathsf{Unused}$ otherwise $r * s = \sqcup$. This allows the following well-typed program, eliminating a pair of Lo and Hi security values, picking the left one to pass to a continuation expecting a Lo input:

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

Metatheory The admissibility of substitution is a key result that holds for this language [32], which is leveraged in soundness of the synthesis calculi.

Lemma 1 (Admissibility of substitution). *Let $\Delta \vdash t' : A$, then:*

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

3 The Synthesis Calculi

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

3.1 Subtractive Resource Management

Our subtractive approach follows the philosophy of earlier work on linear logic proof search [7, 20], structuring synthesis rules around an input context of the available resources and an output context of the remaining resources that can be used to synthesise subsequent subterms. Synthesis rules are read bottom-up, with judgments $\Gamma \vdash A \Rightarrow^{-} t;\, \Delta$ meaning from the *goal type* A we can synthesise a term t using assumptions in Γ, with output context Δ. We describe the rules in turn to aid understanding. The appendix [22] collects the rules for reference.

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

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

On the left, a variable x may be synthesised for the goal A if a linear assumption $x : A$ is present in the input context. The input context without x is then returned as the output context, since x has been used. On the right, we can synthesise a variable x for A we have a graded assumption of x matching the type. However, the grading r must permit x to be used once here. Therefore, the premise states that there exists some grade s such that grade r approximates $s+1$. The grade s represents the use of x in the rest of the synthesised term, and thus $x : [A]_s$ is in the output context. For the natural numbers semiring, this constraint is satisfied by $s = r - 1$ whenever $r \neq 0$, e.g., if $r = 3$ then $s = 2$. For intervals, the role of approximation is more apparent: if $r = [0...3]$ then this rule is satisfied by $s = [0...2]$ where $s+1 = [0...2]+[1...1] = [1...3] \sqsubseteq [0...3]$. Thus, this premise constraint avoids the need for an additive inverse. In the implementation, the constraint is discharged via an SMT solver, where an unsatisfiable result terminates this branch of synthesis.

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

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

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

The left-rule for linear function types then synthesises applications (as in [20]):

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

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

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

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

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

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

$$\frac{\Gamma \vdash A \Rightarrow^- t;\ \Delta}{\Gamma \vdash \Box_r A \Rightarrow^- [t];\ \Gamma - r * (\Gamma - \Delta)}\ \mathrm{R}\Box^-$$

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

Definition 4 (Context subtraction). *For all Γ_1, Γ_2 where $\Gamma_2 \subseteq \Gamma_1$:*

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

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

then $q = [2...2]$ instead of, say, $[0...1]$. This *maximality* condition is important for soundness (that synthesised programs are well-typed).

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

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

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

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

The right and left rules for products, units, and sums, are then fairly straightforward following the subtractive resource model:

$$\frac{\Gamma \vdash A \Rightarrow^- t_1;\ \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow^- t_2;\ \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^- \langle t_1, t_2 \rangle;\ \Delta_2}\ \text{R}\otimes^-$$

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^- t_2;\ \Delta \qquad x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^- \textbf{let}\ \langle x_1, x_2 \rangle = x_3\ \textbf{in}\ t_2;\ \Delta}\ \text{L}\otimes^-$$

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

$$\frac{\Gamma \vdash A \Rightarrow^- t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow^- \textbf{inl}\,t;\ \Delta} \quad \frac{\Gamma \vdash B \Rightarrow^- t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow^- \textbf{inr}\,t;\ \Delta}\ \text{R}\oplus^-_2$$

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

The L$\oplus^-$ rule synthesises the left and right branches of a case statement that may use resources differently. The output context therefore takes the *greatest lower bound* ($\sqcap$) of Δ_1 and Δ_2. We elide definition of context $\sqcap$ as it has the same shape as $\sqcup$ for contexts (Definition 2), just replacing $\sqcup$ with $\sqcap$ on grades.

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

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

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

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

Lemma 2 (Subtractive synthesis soundness). *For all Γ and A then:*

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

i.e. t has type A under context $\Gamma - \Delta$, that contains just those linear and graded variables with grades reflecting their use in t. The appendix [22] provides the proof.

3.2 Additive Resource Management

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

We unpack the rules, starting with variables:

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

For a linear assumption, the output context contains just the variable that was synthesised. For a graded assumption $x : [A]_r$, the output context contains the assumption graded by 1. To synthesise a variable from a graded assumption, we must check that the use is compatible with the grade. The subtractive approach handled this rule ($\textsc{GrVar}^{-}$) by a constraint $\exists s.\, r \sqsupseteq s + 1$. Here however, the point at which we check that a graded assumption has been used according to the grade takes place in the $\mathrm{L}\Box^{+}$ rule, where graded assumptions are bound:

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

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

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

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

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

Synthesising an abstraction (R$\multimap^+$) requires that $x : A$ is in the output context of the premise, ensuring that linearity is preserved. Likewise for application (L$\multimap^+$), the output context of the first premise must contain the linearly bound $x_2 : B$ and the final output context must contain the assumption being used in the application $x_1 : A \multimap B$. This output context computes the *context addition* (Def. 1) of both output contexts of the premises $\Delta_1 + \Delta_2$. If Δ_1 describes how assumptions were used in t_1 and Δ_2 respectively for t_2, then the addition of these two contexts describes the usage of assumptions for the entire subprogram. Recall, context addition ensures that a linear assumption may not appear in both Δ_1 and Δ_2, preventing us from synthesising terms that violate linearity.

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

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

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

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

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

As in the subtractive approach, the right and left rules for products, units, and sums follow fairly straightforwardly from the resource scheme:

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

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

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

$$\frac{\Gamma \vdash A \Rightarrow^+ t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inl}\ t;\ \Delta}\ \mathrm{R}\oplus_1^+ \qquad \frac{\Gamma \vdash B \Rightarrow^+ t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inr}\ t;\ \Delta}\ \mathrm{R}\oplus_2^+$$

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

Rule (L$\oplus^+$) takes the least-upper bound of the premise's output contexts (Def. 2).

Lemma 3 (Additive synthesis soundness). *For all Γ and A:*

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

The appendix [22] provides the proof.

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

$$\dfrac{\Gamma \vdash A \Rightarrow^{+} t_1;\, \Delta_1 \qquad \Gamma - \Delta_1 \vdash B \Rightarrow^{+} t_2;\, \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{+} \langle t_1, t_2\rangle;\, \Delta_1 + \Delta_2}\ \mathrm{R}'\!\otimes^{+}$$

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

Lemma 4 (Additive pruning synthesis soundness). *For all Γ and A:*

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

The appendix [22] provides the proof.

3.3 Focusing

The two calculi provide a foundation for a synthesis algorithm. However, in their current forms, both synthesis calculi are highly non-deterministic: for each rule there are multiple rules which may be applied to synthesise the premise(s).

We apply the idea of *focusing* [3] to derive two *focusing calculi* which are equivalent to the former in expressivity, but with a reduced degree of non-determinism in the rules that may be applied. Focusing is a proof search technique based on the idea that some rules are *invertible*, i.e. whenever the premises of a rule are derivable, the conclusion is also derivable. Rules with this property can be applied eagerly in the synthesis of a term. When we arrive at a goal whose applicable rules are not invertible, we *focus* on either the goal type or a particular assumption by applying a chain of non-invertible rules until we reach a goal to which invertible rules can be applied. The appendix [22] gives focusing versions of the two calculi, which form the basis of our implementation. The proofs for the soundness of these focusing calculi can also be found in the appendix.

4 Evaluation

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

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

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

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

– **Hilbert**: the Hilbert-style axioms of intuitionistic logic (including SKI combinators), with appropriate $\mathbb{N}$ and $\mathbb{N}$-interval grades where needed (see, e.g., S combinator in Example 1 or coproduct elimination in Example 2).

– **Comp**: various translations of function composition into linear logic: multiplicative, call-by-value and call-by-name using ! [17], I/O using ! [28], and coKleisli composition over $\mathbb{N}$ and arbitrary semirings: e.g. $\forall r, s \in \mathcal{R}$:

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

– **Dist**: distributive laws of various graded modalities over functions, sums, and products [23], e.g., $\forall r \in \mathbb{N}$, or $\forall r \in \mathcal{R}$ in any semiring, or $r = [0...\infty]$:

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

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

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

– **Misc**: includes Example 3 (information-flow security) and functions which must share or split resources between graded modalities, e.g.:

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

The appendix [22] lists the type schemes for these synthesis problems (32 in total). We found that Z3 is highly variable in its solving time, so timing measurements are computed as the mean of 20 trials. We used Z3 version 4.8.8 on a Linux laptop with an Intel i7-8665u @ 4.8 Ghz and 16 Gb of RAM.

			Additive			Additive (pruning)			Subtractive	
	Problem		μT (ms)	N		μT (ms)	N		μT (ms)	N
Hilbert	$\otimes$Intro	✓	0.09 (0.00)	0	✓	0.06 (0.01)	0	✓	0.05 (0.00)	0
	$\otimes$Elim	✓	6.23 (0.07)	2	✓	15.05 (0.96)	2	✓	14.27 (0.62)	2
	$\oplus$Intro	✓	0.10 (0.00)	0	✓	0.11 (0.00)	0	✓	0.13 (0.00)	0
	$\oplus$Elim	✓	6.32 (0.22)	2	✓	7.89 (0.26)	2	✓	198.58 (7.01)	15
	SKI	✓	6.38 (0.06)	2	✓	29.39 (0.96)	2	✓	50.59 (5.63)	3
Comp	0/1	✓	32.48 (2.94)	5	✓	41.07 (0.39)	5	×	Timeout	-
	CBN	✓	11.42 (0.35)	3	✓	21.97 (0.60)	3	×	Timeout	-
	CBV	✓	14.57 (0.27)	5	✓	18.60 (0.73)	5	×	Timeout	-
	$\circ coK_{\mathbb{R}}$	✓	21.43 (0.50)	2	✓	25.22 (1.72)	2	×	95.51 (0.94)	8
	$\circ coK_{\mathbb{N}}$	✓	26.04 (1.14)	2	✓	23.70 (1.49)	2	×	101.58 (2.86)	8
	mult	✓	0.14 (0.01)	0	✓	0.13 (0.00)	0	✓	0.15 (0.00)	0
Dist	$\otimes$-!	✓	7.50 (0.21)	2	✓	46.48 (2.68)	2	✓	10482.40 (3.36)	7
	$\oplus$-$\mathbb{N}$	✓	26.43 (0.75)	2	✓	38.33 (2.46)	2	×	15.87 (0.60)	1
	$\otimes$-$\mathbb{N}$	✓	28.18 (0.52)	2	×	35.64 (0.15)	2	×	31.43 (1.21)	2
	$\oplus$-!	✓	11.41 (0.48)	2	✓	13.22 (0.08)	2	✓	165.86 (4.30)	4
	$\otimes$-$\mathcal{R}$	✓	19.35 (0.18)	2	×	23.47 (0.64)	2	×	28.44 (0.80)	2
	$\oplus$-$\mathcal{R}$	✓	20.39 (0.32)	2	✓	23.05 (0.66)	2	×	13.88 (0.16)	1
	$\multimap$-!	✓	10.48 (0.59)	2	✓	12.09 (0.09)	2	✓	344.01 (7.05)	4
	$\multimap$-$\mathbb{N}$	✓	29.22 (1.69)	2	✓	22.02 (0.78)	2	×	64.04 (1.40)	4
	$\multimap$-$\mathcal{R}$	✓	20.20 (0.29)	2	✓	26.75 (0.78)	2	×	54.20 (2.22)	4
Vec	vec5	✓	5.29 (0.14)	1	✓	24.02 (0.81)	1	✓	91.15 (5.07)	6
	vec10	✓	5.46 (0.10)	1	✓	23.74 (3.49)	1	✓	118.95 (1.72)	11
	vec15	✓	9.12 (0.14)	1	✓	23.07 (0.35)	1	✓	181.05 (2.78)	16
	vec20	✓	12.45 (0.05)	1	✓	28.95 (0.10)	1	✓	264.21 (13.54)	21
Misc	split$\oplus$	✓	3.21 (0.05)	1	✓	3.53 (0.10)	1	✓	10715.66 (3.05)	6
	split$\otimes$	✓	9.23 (0.05)	3	✓	43.55 (0.26)	3	×	Timeout	-
	share	✓	268.95 (21.06)	44	✓	112.03 (8.13)	6	✓	190.07 (3.68)	17
	Exm. 3	✓	5.83 (0.14)	2	✓	33.78 (0.79)	2	✓	287.57 (1.56)	3

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

Results and Analysis For each synthesis problem, we recorded whether synthesis was successful or not (denoted ✓ or ×), the mean total synthesis time (μT), the mean total time spent by the SMT solver (μSMT), and the number of calls made to the SMT solver (N). Table 1 summarises the results with the fastest case for each benchmark highlighted. For all benchmarks that used the SMT solver, the solver accounted for $91.73\% - 99.98\%$ of synthesis time, so we report only the mean total synthesis time μT. We set a timeout of 120 seconds.

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

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

In more complex examples we see evidence to support our second hypothesis. The *share* problem requires a lot of graded variable synthesis which is problematic for the additive approach, for the reasons described in the second hypothesis. In contrast, the subtractive approach performs better, with $\mu T = 190.07ms$ as opposed to additive's $268.95ms$. However, additive pruning outperforms both.

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

Overall, the additive approach outperforms the subtractive and is successful at synthesising more examples, including ones polymorphic in grades and even the semiring itself. Given that the literature on linear logic theorem proving is typically subtractive, this is an interesting result. Going forward, a mixed approach between additive and additive pruning may be possible, selecting the algorithm, or even the rules, depending on the class of problem. Exploring this, and further optimisations and improvements, is further work.

Additional Implementation Details Constraints on resource usage are handled via Granule's existing symbolic engine, which compiles constraints on grades (for various semirings) to the SMT-lib format for Z3 [30]. We use the LogicT monad for backtracking search [26] and the Scrap Your Reprinter library for splicing synthesised code into syntactic "holes", preserving the rest of the program text [10]. The implementation of the rule for additive dereliction ($\textsc{der}^{+}$) requires some care. A naïve implementation of this rule would allow the construction of an infinite chain of dereliction applications, by repeatedly applying the rule to the same graded assumption, as the correct usage of the assumption's grade is only verified after it has been used to synthesise a sub-term. Our solution is to simply disallow immediate consecutive applications of the dereliction rule in additive synthesis, requiring that another rule be applied between multiple applications of the dereliction rule to any assumption. If no other rules can be applied, then the branch of synthesis is terminated.

5 Discussion

Further Related Work Before Hodas and Miller [20], the problem of resource non-determinism was first identified by Harland and Pym [19]. Their solution delays splitting of contexts at a multiplicative connective. They later explored the implementation details of this approach, proposing a solution where proof search is formulated in terms of constraints on propositions. The logic programming language Lygon [1] implements this approach.

Our approach to synthesis implements a *backward* style of proof search: starting from the goal, recursively search for solutions to subgoals. In contrast to this, *forward* reasoning approaches attempt to reach the goal by building subgoals from previously proved subgoals until the overall goal is proved. Pfenning and Chaudhuri consider forward approaches to proof search in linear logic using the *inverse method* [11] where the issue of resource non-determinism that is typical to backward approaches is absent [8, 9].

Non-idempotent intersection types systems have a similar core structure resembling the linear λ-calculus with quantitative aspects akin to grading [6]. It therefore seems likely that the approaches of this paper could be applied in this setting and used, for example, as way to enhance or even improve existing work on the inhabitation problem for non-idempotent intersection types [5]: a synthesised term gives a proof of inhabitation. This is left as further work, including formalising the connection between non-idempotent intersections and grading.

Next Steps and Conclusions Our synthesis algorithms are now part of the Granule toolchain with IDE support, allowing programmers to insert a "hole" in a term and, after executing a keyboard shortcut, Granule tries to synthesise the type of the hole, pretty-printing generated code and inserting it at the cursor.

There are various extensions which we are actively pursuing, including synthesis for arbitrary user-defined indexed data types (GADTs), polymorphism, and synthesis of recursive functions. We plan to study various optimisations to the approaches considered here, as well as reducing the overhead of starting the SMT solver each time by instead running an "online" SMT solving procedure. We also plan to evaluate the approach on the extended linear logical benchmarks of Olarte et al. [31]. Although our goal is to create a practical program synthesis tool for common programming tasks rather than a general purpose proof search tool, the approach here also has applications to automated theorem proving.

Acknowledgements Thanks to Benjamin Moon, Harley Eades III, participants at LOPSTR 2020, and the anonymous reviewers for their helpful comments. This work is supported by an EPSRC Doctoral Training Award and EPSRC grant EP/T013516/1 (*Verifying Resource-like Data Use in Programs via Types*).

References

1. Logic programming with linear logic, http://www.cs.rmit.edu.au/lygon/., accessed 19th June 2020
2. 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)
3. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation **2**(3), 297–347 (06 1992). https://doi.org/10.1093/logcom/2.3.297
4. 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
5. Bucciarelli, A., Kesner, D., Rocca, S.R.D.: Inhabitation for non-idempotent intersection types. Log. Methods Comput. Sci. **14**(3) (2018). https://doi.org/10.23638/LMCS-14(3:7)2018, https://doi.org/10.23638/LMCS-14(3:7)2018
6. Bucciarelli, A., Kesner, D., Ventura, D.: Non-idempotent intersection types for the lambda-calculus. Log. J. IGPL **25**(4), 431–464 (2017). https://doi.org/10.1093/jigpal/jzx018, https://doi.org/10.1093/jigpal/jzx018
7. Cervesato, I., Hodas, J.S., Pfenning, F.: Efficient resource management for linear logic proof search. Theoretical Computer Science **232**(1), 133 – 163 (2000). https://doi.org/https://doi.org/10.1016/S0304-3975(99)00173-5
8. Chaudhuri, K., Pfenning, F.: A Focusing Inverse Method Theorem Prover for First-Order Linear Logic. In: Proceedings of the 20th International Conference on Automated Deduction. p. 69–83. CADE' 20, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11532231_6
9. Chaudhuri, K., Pfenning, F.: Focusing the Inverse Method for Linear Logic. In: Proceedings of the 19th International Conference on Computer Science Logic. p. 200–215. CSL'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11538363_15
10. Clarke, H., Liepelt, V.B., Orchard, D.: Scrap your Reprinter (2017), unpublished manuscript
11. Degtyarev, A., Voronkov, A.: Chapter 4 - the inverse method. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 179 – 272. Handbook of Automated Reasoning, North-Holland, Amsterdam (2001). https://doi.org/https://doi.org/10.1016/B978-044450813-3/50006-0
12. Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. ACM SIGPLAN Notices **51**(1), 802–815 (2016)
13. Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A., Pierce, B.C.: Linear dependent types for differential privacy. SIGPLAN Not. **48**(1), 357–370 (Jan 2013). https://doi.org/10.1145/2480359.2429113
14. 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 (2016). https://doi.org/10.1145/2951913.2951939

15. Gentzen, G.: Untersuchungen über das logische schließen. ii. Mathematische Zeitschrift **39**, 405–431 (1935)
16. 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
17. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1 – 101 (1987). https://doi.org/https://doi.org/10.1016/0304-3975(87)90045-4
18. 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)
19. Harland, J., Pym, D.J.: Resource-distribution via boolean constraints. CoRR **cs.LO/0012018** (2000), `https://arxiv.org/abs/cs/0012018`
20. Hodas, J., Miller, D.: Logic Programming in a Fragment of Intuitionistic Linear Logic. Information and Computation **110**(2), 327 – 365 (1994). https://doi.org/https://doi.org/10.1006/inco.1994.1036
21. Howard, W.A.: The Formulae-as-types Notion of Construction. In: Seldin, J.P., Hindley, J.R. (eds.) To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. Academic Press (1980)
22. Hughes, J., Orchard, D.: Resourceful Program Synthesis from Graded Linear Types (Appendix) (2020). https://doi.org/10.5281/zenodo.4314644, `https://doi.org/10.5281/zenodo.4314644`
23. Hughes, J., Vollmer, M., Orchard, D.: Deriving distributive laws for graded linear types (2020), unpublished manuscript
24. 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
25. Katsumata, S.: A double category theoretic analysis of graded linear exponential comonads. In: Baier, C., Lago, U.D. (eds.) Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Thessaloniki, Greece, April 14-20, 2018. Lecture Notes in Computer Science, vol. 10803, pp. 110–127. Springer (2018). https://doi.org/10.1007/978-3-319-89366-2_6
26. Kiselyov, O., Shan, C.c., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers: (functional pearl). SIGPLAN Not. **40**(9), 192–203 (Sep 2005). https://doi.org/10.1145/1090189.1086390
27. Knoth, T., Wang, D., Polikarpova, N., Hoffmann, J.: Resource-Guided Program Synthesis. CoRR **abs/1904.07415** (2019), `http://arxiv.org/abs/1904.07415`
28. Liang, C., Miller, D.: Focusing and polarization in linear, intuitionistic, and classical logics. Theoretical Computer Science **410**(46), 4747–4768 (2009)
29. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Transactions on Programming Languages and Systems (TOPLAS) **2**(1), 90–121 (1980)
30. de Moura, L., Bjørner, N.: Z3: an efficient smt solver. vol. 4963, pp. 337–340 (04 2008)
31. Olarte, C., de Paiva, V., Pimentel, E., Reis, G.: The ILLTP library for intuitionistic linear logic. In: Ehrhard, T., Fernández, M., de Paiva, V., de Falco, L.T. (eds.) Proceedings Joint International Workshop on Linearity & Trends in Linear

Logic and Applications, Linearity-TLLA@FLoC 2018, Oxford, UK, 7-8 July 2018. EPTCS, vol. 292, pp. 118–132 (2018). https://doi.org/10.4204/EPTCS.292.7
32. 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
33. Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. SIGPLAN Not. **50**(6), 619–630 (Jun 2015). https://doi.org/10.1145/2813885.2738007
34. Polikarpova, N., Solar-Lezama, A.: Program synthesis from Polymorphic Refinement Types. CoRR **abs/1510.08419** (2015), `http://arxiv.org/abs/1510.08419`
35. Smith, C., Albarghouthi, A.: Synthesizing differentially private programs. Proc. ACM Program. Lang. **3**(ICFP) (Jul 2019). https://doi.org/10.1145/3341698
36. Zalakain, U., Dardha, O.: Pi with leftovers: a mechanisation in Agda. arXiv preprint arXiv:2005.05902 (2020)