Quantitative Polynomial Functors

Georgi Nakov
Department of Computer and Information Sciences, University of Strathclyde, U.K.

Fredrik Nordvall Forsberg
Department of Computer and Information Sciences, University of Strathclyde, U.K.

Abstract

We investigate containers and polynomial functors in Quantitative Type Theory, and give initial algebra semantics of inductive data types in the presence of linearity. We show that reasoning by induction is supported, and equivalent to initiality, also in the linear setting.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Linear logic

Keywords and phrases quantitative type theory, polynomial functors, inductive data types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.9

Supplementary Material Idris 2 formalisation: `https://github.com/g-nakov/quantitative-poly`

1 Introduction

Data types are the basic building blocks of modern type theories and programming languages. Having more powerful data types around can increase the proof-theoretic strength of the theory [34], i.e., allow more programs to be written, and can also make existing proofs/programs more convenient to write [13]. Recent advances in type theories such as cubical type theory [11] have also been accompanied by advances in data type theory, such as quotient and higher inductive types [5, 12, 30]. In this paper, we explore what a corresponding notion of (non-higher, so far) inductive types for the also recently introduced type theory Quantitative Type Theory (QTT) [6, 31] might be. QTT combines dependent types and linear types, in the sense of linear logic [20, 37]. By using linearity to track variable (and hence resource) usage of programs, QTT thus promises to enable formal reasoning about both functional and non-functional correctness of programs. A variant of QTT is implemented in the Idris 2 programming language [8], and we hope that our work can be used as a foundational justification for the implementation of data types there. Conversely, we have used Idris 2 to mechanically verify parts of our development.

In a linear world, there are exciting new data types to explore, such as binary trees where only one subtree at each node is present at runtime. Such trees could be used to reduce memory usage and network traffic, or to accurately model the situation at hand. For example, if the tree represents an I/O program in the sense of Hancock and Setzer [22], where subtrees stand for continuation computations, then we would like to ensure that indeed only one continuation is invoked — we do not want the user to simultaneously launch the nukes, and refrain from doing so, by exploring two different subtrees!

While it would be possible to add such data types to QTT by manually writing down formation, introduction, elimination and computation rules for them in an ad-hoc fashion, this is a cumbersome and error-prone process. The goal of this paper is to present a more principled solution. We would like to *derive* the rules for data types from some kind of canonical rules. Our instinct is to turn to the theory of polynomial functors [18, 19], also known as containers [1]. These specify data types as given by shapes and positions, with a data value given by choosing a shape, and then filling every position of the data type with a payload. Pleasingly, containers are closed under many constructions on types such as products and coproducts, and so all strictly positive data types can be reduced to (fixed points of) containers in “traditional” type theory [2, 15, 25].

Hence we present a quantitative version of containers, replacing dependent pair types with dependent tensor types, and function spaces with linear function spaces. Fixed points of such containers now more or less immediately admit formation and introduction rules for their corresponding data types, but what about the elimination and computation rules? Our main technical contribution is to show that the problem of providing an elimination rule for fixed points of such containers can be reduced to the problem of proving initiality in a category of algebras. For traditional data types, this is well known [23, 7], and our proof is a linear refinement of the traditional proof — pairs with “unused” first components play a key role in the construction.

Is that the end of the story? Unfortunately not! Ordinary containers are useful as a foundation for data types in traditional type theory exactly since all strictly positive data types can be reduced to them. However when we try to replicate this reduction for quantitative data types, we hit a snag: in the quantitative setting, it is no longer the case that every strictly positive type is isomorphic to a quantitative container, because of the splitting up of connectives into additive and multiplicative variants. Quantitative containers still serve as an instructive special case of quantitative data types, but they do not cover every interesting example. Instead, we define quantitative polynomial functors as inductively generated from a grammar containing constants, the identity, and sums and products. We can show that also for this class of functors, elimination rules are supported if and only if the algebra is initial. The proof is similar to the proof for quantitative containers, but now further complicated by having to also do induction on the generation according to the polynomial functors grammar.

Our second major contribution is to show that finitary polynomial functors indeed have initial algebras in Atkey’s linear realisability model of QTT [6, § 4]. This is a model where types are interpreted as *assemblies*, i.e., sets whose elements are assigned realisers from a combinatory algebra of (untyped) linear programs. We construct the initial algebras in two stages: first we consider the initial algebra of the polynomial functor in the category $\mathcal{S}et$, which is known to exist. We then use the initiality of the algebra in $\mathcal{S}et$ to both define the realisability relation, and to prove that the algebra map and mediating morphism into other algebras are realised — this constructs an initial algebra also in the category of assemblies. Putting our contributions together, we have thus defined a syntactic class of polynomial data types, and shown that QTT extended with rules for them, including dependent elimination rules, can be soundly interpreted in Atkey’s realisability model.

Structure of the paper

In Section 2, we recall the syntax and semantics of QTT, including a sketch of a realisability model. In Section 3, we review the traditional theory of initial algebras and containers. We then move from containers to quantitative containers in Section 4, and prove that initiality is equivalent to dependent elimination for them. In Section 5, we generalise quantitative containers to inductively generated quantitative polynomial functors, and show that also for them, initiality is equivalent to dependent elimination. We finally construct initial algebras of finitary quantitative polynomial functors in the aforementioned realisability model.

Partial Idris 2 formalisation

We have formalised the basic definitions and results from Section 4 and Section 5 in Idris 2, most notably including a verification of Theorem 16 and the induction-from-initiality direction of Theorem 26. We make use of Idris 2's implementation of linearity to faithfully model quantitative containers and quantitative polynomial functors, and to verify that the morphisms we construct, such as dist_F from Lemma 24, really are linear. It was helpful to have Idris 2 guide us by the quantity information *during* the construction of the proofs — for example, feedback from Idris 2 made us realise the correct definition of lifting of constant functors in Definition 20.

The code can be found at `https://github.com/g-nakov/quantitative-poly`.

2 Quantitative Type Theory

In this section, we give a brief introduction to the syntax and semantics of QTT, and present the typing rules for the type formers we will use. For a detailed presentation, see Atkey [6]. In contrast to dependent linear/non-linear type theories [9, 29], QTT maintains a single context in which variables can both contribute to type formation, and be marked as a linear resource. This is in line with recent work by Fu, Kishida and Selinger [17], and Abel and Bernardy [3]. See also Choudhury, Eades III, Eisenberg and Weirich [10], and Orchard, Liepelt and Eades III [32] for similar approaches.

2.1 Syntax of Quantitative Type Theory

The main difference between quantitative and ordinary type theory is that variables in quantitative type theory contexts are annotated with resources, intuitively governing how many times they can be used. These resource annotations are drawn from a semiring, satisfying some additional axioms that are needed for the typing rules to make sense.

▶ **Definition 1.** *A* resource semiring *is a structure* $R = \langle R, +, \cdot, 0, 1\rangle$*, such that* $\langle R, +, 0\rangle$ *is a monoid,* $\langle R, \cdot, 1\rangle$ *is a commutative monoid, with* $\cdot$ *distributing over* $+$*, such that for every* $\rho, \pi \in R$*,* $\rho \cdot \pi = 0$ *if and only if* $\rho = 0$ *or* $\pi = 0$*, and such that* $\rho + \pi = 0$ *implies* $\rho = \pi = 0$*.*

Examples of resource semirings are given by the natural numbers with ordinary addition and multiplication, and the "zero-one-many" resource semiring $\langle\{0, 1, \omega^<\}, +, \cdot, 0, 1\rangle$ with $x + \omega^< = \omega^< + x = \omega^<$, $1 + 1 = \omega^<$, and $\omega^< \cdot \omega^< = \omega^<$.

Fixing a semiring R, a QTT term judgement has the following form:

$$x_1 \overset{\pi_1}{:} S_1, x_2 \overset{\pi_2}{:} S_2, \dots x_n \overset{\pi_n}{:} S_n \vdash M \overset{\sigma}{:} T$$

with $\pi_i \in R$ and σ restricted to either $\sigma = 0_R$ or $\sigma = 1_R$. The list of variables x_i on the left of the turnstile forms the context of the judgement. The resource semiring operations can be lifted pointwise to contexts:

$$\pi(\Gamma, x \overset{\rho}{:} S) = \pi\Gamma, x \overset{\pi\rho}{:} S$$
$$(\Gamma_1, x \overset{\rho_1}{:} S) + (\Gamma_2, x \overset{\rho_2}{:} S) = (\Gamma_1 + \Gamma_2), x \overset{\rho_1+\rho_2}{:} S, \text{ if } 0\Gamma_1 = 0\Gamma_2.$$

In typical dependent type theory fashion, the variables x_i are available for forming both types and terms. Each variable x_i is annotated with a resource π_i denoting how many times x_i must be used *computationally* in the term M on the right hand side of the turnstile. In contrast, the calculus is set up so that types are always formed in contexts of the form 0Γ —

$$\frac{}{0\Gamma \vdash \mathbf{I} : \mathsf{Type}} \quad \frac{}{0\Gamma \vdash \top : \mathsf{Type}} \quad \frac{}{0\Gamma \vdash \mathbf{0} : \mathsf{Type}}$$

$$\frac{}{0\Gamma \vdash \star : \mathbf{I}} \quad \frac{0\Gamma_1, x \overset{0}{:} \mathbf{I} \vdash S : \mathsf{Type} \quad \Gamma_1 \vdash m : \mathbf{I} \quad \Gamma_2 \vdash n : S[\star/x]}{\Gamma_1 + \Gamma_2 \vdash \text{ let } \star = m \text{ in } n : S[m/x]} \quad \frac{}{\Gamma \vdash \star : \top} \quad \frac{\Gamma \vdash m : \mathbf{0}}{\Gamma \vdash \mathsf{abort}\ m : B}$$

Figure 1 Type formation and rules for basic types

type formation consume no resources. The distinction between computational and "vacuous" usage is marked explicitly by the annotation σ on the term M in the conclusion, which can only be 0 or 1. Such a restriction is needed to retain admissibility of substitution and effectively splits the theory in two fragments. Terms in the 0 fragment bear no computational content, consume no resources, and are available at any point in a derivation, while the inhabitants of the 1 fragment are computationally relevant, but need sufficient resources to be constructed. The following meta-result, which states that "zero needs nothing", is useful for constructing derivations in the 0 fragment, basically without worrying about resource usage:

▶ **Lemma 2** (Atkey [6]). *If $\Gamma \vdash M \overset{0}{:} S$, then $\Gamma = 0\Gamma$.* ◀

From now on, we follow the following conventions: Types are formed in the 0 fragment, but we may omit the 0 annotation, that is, we may write $0\Gamma \vdash S : \mathsf{Type}$ for $0\Gamma \vdash S \overset{0}{:} \mathsf{Type}$. Any other judgement takes place in the 1 fragment, unless explicitly annotated otherwise, and we suppress the annotation on the conclusion — that is, $\Gamma \vdash M : S$ should be read as $\Gamma \vdash M \overset{1}{:} S$. Scaling the context by 0 yields the corresponding rule in the 0 fragment. We further assume that contexts are well formed; for example, we only consider the context $\Gamma_1 + \Gamma_2$ if $0\Gamma_1 = 0\Gamma_2$.

As an example, consider the type family $\mathsf{Fin} : \mathbb{N} \to \mathsf{Type}$ of finite types. It has formation rule

$$\frac{0\Gamma \vdash n \overset{0}{:} \mathbb{N}}{0\Gamma \vdash \mathsf{Fin}(n) : \mathsf{Type}}$$

where we ask for 0 copies of $n : \mathbb{N}$, since this n occurs in a type. The introduction rules are

$$\frac{0\Gamma \vdash n \overset{0}{:} \mathbb{N}}{0\Gamma \vdash \mathsf{zero} : \mathsf{Fin}(n+1)} \qquad \frac{0\Gamma \vdash n \overset{0}{:} \mathbb{N} \qquad \Gamma \vdash m : \mathsf{Fin}(n)}{\Gamma \vdash \mathsf{suc}(m) : \mathsf{Fin}(n+1)}$$

where again $n : \mathbb{N}$ does not require any resources, since it occurs only in a type. We omit the elimination rule here, since we will not make use of it further. Rules for other basic types can be found in Figure 1; the type $\mathbf{I}$ was considered by Atkey [6, §2.1.3], whereas $\top$ and $\mathbf{0}$ are first described for QTT here. The monoidal unit type $\mathbf{I}$ contains no information, and hence it is always possible to construct an element $\star : \mathbf{I}$ at no cost, or use the elimination rule to discard such an element. The terminal type $\top$ superficially looks very similar to the monoidal unit $\mathbf{I}$, but note that it is possible to construct $\star : \top$ (we reuse the same syntax $\star$ for the term as for $\mathbf{I}$) even in non-zero contexts, which makes it terminal. There is no elimination rule. Dually the empty type $\mathbf{0}$ allows elimination into any other type.

Rules for type formers are given in Figure 2; $(x \overset{\rho}{:} S) \otimes T$ and $(x \overset{\rho}{:} S) \to T$ were considered by Atkey, and $(x : S) \,\&\, T$ was introduced by Svoboda [35], whereas $S \oplus T$ is new to QTT, as

$$\frac{0\Gamma \vdash S : \mathsf{Type} \quad 0\Gamma, x \overset{0}{:} S \vdash T : \mathsf{Type}}{0\Gamma \vdash (x \overset{\rho}{:} S) \to T : \mathsf{Type}} \qquad \frac{\Gamma, x \overset{\rho}{:} S \vdash t : T}{\Gamma \vdash \lambda x.\, t : (x \overset{\rho}{:} S) \to T}$$

$$\frac{\Gamma_1 \vdash f : (x \overset{\rho}{:} S) \to T \quad \Gamma_2 \vdash s : S}{\Gamma_1 + \rho\Gamma_2 \vdash f(s) : S[s/x]}$$

$$\frac{0\Gamma \vdash S : \mathsf{Type} \quad 0\Gamma, x \overset{0}{:} S \vdash T : \mathsf{Type}}{0\Gamma \vdash (x \overset{\rho}{:} S) \otimes T : \mathsf{Type}} \qquad \frac{\Gamma_1 \vdash s : S \quad \Gamma_2 \vdash t : T[s/x]}{\rho\Gamma_1 + \Gamma_2 \vdash (s, t) : (x \overset{\rho}{:} S) \otimes T}$$

$$\frac{0\Gamma_1, z \overset{0}{:} (x \overset{\rho}{:} S) \otimes T \vdash U : \mathsf{Type} \quad \Gamma_1 \vdash p : (x \overset{\rho}{:} S) \otimes T \quad \Gamma_2, x \overset{\rho}{:} S, y \overset{1}{:} T \vdash u : U[(x,y)/z]}{\Gamma_1 + \Gamma_2 \vdash \text{ let } (x, y) = p \text{ in } u : U[p/z]}$$

$$\frac{0\Gamma \vdash S : \mathsf{Type} \quad 0\Gamma, x \overset{0}{:} S \vdash T : \mathsf{Type}}{0\Gamma \vdash (x : S) \mathbin{\&} T : \mathsf{Type}} \qquad \frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash (s, t) : (x : S) \mathbin{\&} T}$$

$$\frac{\Gamma \vdash p : (x : S) \mathbin{\&} T}{\Gamma \vdash \mathsf{proj}_1(p) : S} \qquad \frac{\Gamma \vdash p : (x : S) \mathbin{\&} T}{\Gamma \vdash \mathsf{proj}_2(p) : T[\mathsf{proj}_1(p)/x]}$$

$$\frac{0\Gamma \vdash S : \mathsf{Type} \quad 0\Gamma \vdash T : \mathsf{Type}}{0\Gamma \vdash S \oplus T : \mathsf{Type}} \qquad \frac{\Gamma \vdash s : S}{\Gamma \vdash \mathsf{inl}\, s : S \oplus T} \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash \mathsf{inr}\, t : S \oplus T}$$

$$\frac{0\Gamma_1, z \overset{0}{:} S \oplus T \vdash U : \mathsf{Type} \quad \Gamma_1 \vdash e : S \oplus T \quad \Gamma_2, x \overset{1}{:} S \vdash u : U[\mathsf{inl}\, x/z] \quad \Gamma_2, y \overset{1}{:} T \vdash v : U[\mathsf{inr}\, y/z]}{\Gamma_1 + \Gamma_2 \vdash \text{ case } e \text{ of } \mathsf{inl}\, x \Rightarrow u \mid \mathsf{inr}\, y \Rightarrow v : U[e/z]}$$

$$\frac{0\Gamma \vdash S : \mathsf{Type} \quad 0\Gamma \vdash s \overset{0}{:} S \quad 0\Gamma \vdash t \overset{0}{:} S}{0\Gamma \vdash \mathsf{Id}_S(s, t) : \mathsf{Type}} \qquad \frac{0\Gamma \vdash s = t : S}{0\Gamma \vdash \mathsf{refl} : \mathsf{Id}_S(s, t)} \qquad \frac{0\Gamma \vdash p : \mathsf{Id}_S(s, t)}{0\Gamma \vdash s = t : S}$$

Figure 2 Typing rules for type formers.

far as we know. Extra care should be taken in the rules for types with binders, as the bound variable could potentially represent a computational resource — for example, the dependent function type $(x \overset{\rho}{:} S) \to T$ records how many copies ρ of its argument are needed. We write $S \xrightarrow{\rho} T$ for $(x \overset{\rho}{:} S) \to T$ if x does not occur in T. The dependent tensor type $(x \overset{\rho}{:} S) \otimes T$ is the multiplicative linear version of the dependent pair type; it contains pairs (s, t) with ρ copies of s and one copy of t, which must both be consumed fully in the elimination rule. We pay special attention to that type as a key to a simpler presentation of the remaining rules. Using the monoidal unit $\mathbf{I}$, we can form an exponential type $!_\rho S = (x \overset{\rho}{:} S) \otimes \mathbf{I}$ and "disguise" resource usage information within the types themselves. Thus, we avoid proliferating the typing rules with abundant annotations while still retaining the full expressiveness of QTT. For example, no modifications in the rules are needed to allow for arbitrarily many copies of the second component of the tensor tuple — working with the type $(x \overset{\rho}{:} S) \otimes (!_\pi T)$ already achieves that.

In contrast to the dependent tensor type, the dependent additive conjunction $(x : S) \mathbin{\&} T$ (pronounced "with") contains pairs (s, t) where the consumer can choose to either use $s : S$

$$\frac{\Gamma \vdash n : S[\star/x]}{\Gamma \vdash \text{ let } \star = \star \text{ in } n \equiv n : S[\star/x]} \qquad \frac{\Gamma_1, x \overset{\rho}{:} S \vdash t : T \quad \Gamma_2 \vdash s : S}{\Gamma_1 + \rho\Gamma_2 \vdash (\lambda x.\, t)\, s \equiv t[s/x] : T[s/x]}$$

$$\frac{\Gamma_1 \vdash s : S \quad \Gamma_2 \vdash t : T[s/x] \quad \Gamma_3, x \overset{\rho}{:} S, y \overset{1}{:} T \vdash u : U[(x,y)/z]}{\rho\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \text{ let } (x,y) = (s,t) \text{ in } u \equiv u[s/x, t/y] : U[(s,t)/z]}$$

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash \mathsf{proj}_1(s,t) \equiv s : S} \qquad \frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash \mathsf{proj}_2(s,t) \equiv t : T[s/x]}$$

$$\frac{\Gamma_1 \vdash s : S \quad \Gamma_2, x \overset{1}{:} S \vdash u : U[\mathsf{inl}\, x/z] \quad \Gamma_2, y \overset{1}{:} T \vdash v : U[\mathsf{inr}\, y/z]}{\Gamma_1 + \Gamma_2 \vdash \text{ case } \mathsf{inl}\, s \text{ of } \mathsf{inl}\, x \Rightarrow u \mid \mathsf{inr}\, y \Rightarrow v \equiv u[s/x] : U[\mathsf{inl}\, s/z]}$$

$$\frac{\Gamma_1 \vdash t : T \quad \Gamma_2, x \overset{1}{:} S \vdash u : U[\mathsf{inl}\, x/z] \quad \Gamma_2, y \overset{1}{:} T \vdash v : U[\mathsf{inr}\, y/z]}{\Gamma_1 + \Gamma_2 \vdash \text{ case } \mathsf{inr}\, t \text{ of } \mathsf{inl}\, x \Rightarrow u \mid \mathsf{inr}\, y \Rightarrow v \equiv v[t/y] : U[\mathsf{inr}\, t/z]}$$

$$\frac{\Gamma \vdash t : \top}{\Gamma \vdash t \equiv \star : \top} \qquad \frac{\Gamma \vdash f : (x \overset{\rho}{:} S) \to T}{\Gamma \vdash f \equiv \lambda x.\ f(x) : (x \overset{\rho}{:} S) \to T}$$

Figure 3 β- and η-rules.

or $t : T[s/x]$. We write $S \otimes T$ for $(x \overset{1}{:} S) \otimes T$ and $S \,\&\, T$ for $(x : S) \,\&\, T$ respectively, if x does not occur in T — these types correspond to the usual non-dependent linear logic connectives. Finally $S \oplus T$ is the disjoint union of S and T; note that both branches share the same context Γ_2 in its elimination rule. Combining with exponential types, we can recover the more general version $(!_\rho S) \oplus (!_\pi T)$. We can define the type of Booleans by $\mathbf{2} = \mathbf{I} \oplus \mathbf{I}$. In particular, this implies that the constructors $\mathsf{false} = \mathsf{inl}\, \star$ and $\mathsf{true} = \mathsf{inr}\, \star$ can be introduced using zero resources. To facilitate formal reasoning within QTT, we also add extensional identity types. Because of equality reflection, a term p of type $\mathsf{Id}_S(s,t)$ does not use any computational resources, and thus can be introduced even if the resources designated by the terms s and t have been exhausted. We chose to an extensional identity type since they are easier to work with, and easy to model in realisability models — in particular, with an extensional identity type we do not need to worry about if the rules for the identity type should be formulated multiplicatively, additively, or both. In principle one could also try to develop QTT with an intensional flavour — linearity and identity types appear to be orthogonal features in the type system.

Eliminators come with their expected β- and η-rules familiar from ordinary type theory, listed in Figure 3. Note that we have only included η-rules for $\top$ and dependent function types — all other type formers have eliminators that can be used to derive the expected η-rules using the extensional identity type, but $\top$ does not have an eliminator, and the η-rule for dependent function types is needed to derive function extensionality from the extensional identity type.

These rules give types a rich algebraic structure. An isomorphism of types $A \cong B$ is given by two linear functions $\vdash f : A \overset{1}{\to} B$ and $\vdash g : B \overset{1}{\to} A$ such that $x \overset{0}{:} A \vdash g(f(x)) = x : A$ and $y \overset{0}{:} B \vdash f(g(y)) = y : B$ — using equality reflection, these equalities can also be proven using the extensional identity type. For example, the usual currying-uncurrying isomorphism, relating dependent functions and dependent pairs, becomes $(z \overset{1}{:} ((x \overset{\rho}{:} A) \otimes B) \to C) \cong$

$(x \overset{\rho}{:} A) \to (y \overset{1}{:} B) \to C[(x,y)/z]$ in the QTT setting. We will make use of the following isomorphisms.

▶ **Lemma 3.** *For any type A, we have:*

$$\mathbf{I} \otimes A \cong A \qquad \mathbf{0} \xrightarrow{1} A \cong \top \qquad \mathbf{I} \xrightarrow{1} A \cong A \qquad \mathbf{2} \xrightarrow{1} A \cong A \,\&\, A$$ ◀

In contrast, for example $\top \otimes A \ncong A$ and $(A \xrightarrow{1} \mathbf{I}) \ncong \mathbf{I}$, intuitively because there is no way to consume an element of $\top$, or to produce a function $A \xrightarrow{1} \mathbf{I}$ in general – this would allow discarding elements of A. Formally, one can show that such isomorphisms do not exist by exhibiting concrete models of QTT where they do not hold.

2.2 Semantics of Quantitative Type Theory

Categories with Families [14] (CwFs) form a sound and complete semantics for dependent type theories, and are given by a category $\mathcal{C}$ (modelling contexts and substitutions), together with a functor $(\mathsf{Ty}, \mathsf{Tm}) : \mathcal{C} \to \mathsf{Fam}(\mathcal{S}et)$ into the category of families of sets (modelling types and terms in a given context), and a context comprehension operation $-.- : (\Gamma : \mathcal{C}) \to \mathsf{Ty}(\Gamma) \to \mathcal{C}$ (modelling context extension), with a universal property. Atkey [6] presents a refined notion of *quantitative* CwF to account for QTT's usage information in contexts, terms and types. Intuitively, a quantitative CwF for a fixed resource semiring R is given by *two* categories $\mathcal{L}$ and $\mathcal{C}$, used to model resourced contexts and contexts in the 0 fragment, respectively. To any resourced context corresponds a plain one, obtained by forgetting the resource annotations, and so there should be a faithful functor $U : \mathcal{L} \to \mathcal{C}$. Since the 0 fragment allows unrestricted usage, $\mathcal{C}$ should be a CwF on its own. Types are always formed in the 0 fragment, so there is no need to have a separate notion of "resourced types", but $\mathcal{L}$ needs to support both resourced context extension and resourced terms, both suitably displayed by U over their unresourced counterparts in $\mathcal{C}$. Finally, we need to ask for functors $\rho(-) : \mathcal{L} \to \mathcal{L}$ for each $\rho \in R$, and $(+) : \mathcal{L} \times_{\mathcal{C}} \mathcal{L} \to \mathcal{L}$ (where $\mathcal{L} \times_{\mathcal{C}} \mathcal{L}$ is the pullback of U along itself), modelling scaling and context addition, respectively.

Since quantitative CwFs are refinements of CwFs, we can construct a "trivially quantitative" CwF for each ordinary CwF $\mathcal{C}$ by putting $\mathcal{L} = \mathcal{C}$ with $U = \mathsf{Id}$, and $\mathsf{RTm} = \mathsf{Tm}$. We can also build models that actually demonstrate the distinction between computational and non-computational use of data. Atkey [6] constructs a model based on linear realisability [24], which we now describe, since we will make use of it in what follows. Unresourced terms will be modelled by set-theoretic functions, while their resourced counterparts will be supplemented with computable, linear realisers, drawn from an R-linear combinatory algebra.

▶ **Definition 4.** *An R-*linear combinatory algebra *(R-LCA) consists of a set $\mathcal{A}$, a binary operation $(\cdot) : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$, a family of unary operations $!_\rho : \mathcal{A} \to \mathcal{A}$ for $\rho \in R$, and combinators $B, C, I, K, D, W_{\pi\rho}, \delta_{\pi\rho}, F_\rho \in \mathcal{A}$ for every $\pi, \rho \in R$, satisfying the following equations:*

- $B \cdot x \cdot y \cdot z = x \cdot (y \cdot z)$
- $C \cdot x \cdot y \cdot z = x \cdot z \cdot y$
- $I \cdot x = x$
- $K \cdot x \cdot !_0 y = x$
- $D \cdot !_1 x = x$
- $W_{\pi\rho} \cdot x \cdot !_{\pi+\rho} y = x \cdot !_\pi y \cdot !_\rho y$
- $\delta_{\pi\rho} \cdot !_{\pi\rho} x = !_\pi !_\rho x$
- $F_\rho \cdot !_\rho x \cdot !_\rho y = !_\rho (x \cdot y)$

An R-LCA $\mathcal{A}$ supports booleans *if there exists elements $T, F \in \mathcal{A}$ and a function* $\mathsf{case} : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$*, such that for all $p, q \in \mathcal{A}$, $\mathsf{case}(p,q) \cdot T = p$ and $\mathsf{case}(p,q) \cdot F = q$.*

The operation $\cdot$ is meant to represent function application; the combinators B, C, I are used to implement composition, exchange and identity, $!_\rho$ represents "making ρ copies", with K, D, $W_{\pi\rho}$, $\delta_{\pi\rho}$, and F_ρ implementing structural rules and reshuffling of resources — these witness that $!_{\_} : R \times \mathcal{A} \to \mathcal{A}$ is an action. See Abramsky, Haghverdi and Scott [4], and Hoshino [24] for more details. Importantly, R-linear combinatory algebras are combinatorily complete: given an expression M built from applications, elements of $\mathcal{A}$, and variables, with exactly one occurrence of the variable x, there exists an expression $\lambda^* x.M$, not containing x, such that $(\lambda^* x.M) \cdot N = M[N/x]$. We define the tupling of elements $a_1, \ldots, a_n \in \mathcal{A}$ using combinatory completeness and the standard Church encoding $[x_1, \ldots, x_n] := \lambda^* q.q \cdot x_1 \cdot \ldots x_n$.

▶ **Example 5.** Fix R as the zero-one-many semiring $\{0, 1, \omega^<\}$, and let $\langle -, - \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be a bijection, and $[-] : \mathsf{List}\,\mathbb{N} \to \mathbb{N}$ an encoding of finite lists of natural numbers as natural numbers, in such a way that there is a "list membership" predicate $a \in b$ and a list concatenation $a +\!\!+ b$ such that $a_i \in [a_1, \ldots, a_n]$, and $[a_1, \ldots, a_n] +\!\!+ [a_{n+1}, \ldots, a_m] = [a_1, \ldots, a_m]$ (with no requirements on $a \in bs$ and $as +\!\!+ bs$ when as and bs are not in the image of $[-]$). Following Atkey [6, Examples 4.3, 4,6 and 4.7], who in turn followed Hoshino [24, §5.3], we can endow the power set $\mathcal{P}(\mathbb{N})$ of $\mathbb{N}$ with the structure of an R-LCA by defining the application $\alpha \cdot \beta = \{n \mid \langle m, n \rangle \in \alpha, m \in \beta\}$. We write $[a_1, \ldots, a_\rho]$ for a list of length ρ, i.e. $[a_1, \ldots, a_0] = []$, $[a_1, \ldots, a_1] = [a_1]$ and $[a_1, \ldots, a_{\omega^<}]$ means $[a_1, \ldots, a_n]$ for some $n \in \mathbb{N}$, and we define $!_\rho : \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$ by $!_\rho \alpha = \{[a_1, \ldots, a_\rho] \mid a_i \in \alpha\}$. The combinators are given by:

$$
\begin{aligned}
&B = \{\langle\langle m, k\rangle, \langle\langle n, m\rangle, \langle n, k\rangle\rangle\rangle \mid n, m, k \in \mathbb{N}\}\\
&C = \{\langle\langle m, \langle n, k\rangle\rangle, \langle n, \langle m, k\rangle\rangle\rangle \mid n, m, k \in \mathbb{N}\}\\
&I = \{\langle n, n\rangle \mid n \in \mathbb{N}\}\\
&K = \{\langle n, \langle [], n\rangle\rangle \mid n \in \mathbb{N}\}\\
&D = \{\langle [n], n\rangle \mid n \in \mathbb{N}\}\\
&W_{\pi\rho} = \{\langle\langle a_1, \langle a_2, b\rangle\rangle, \langle a_1 +\!\!+ a_2, b\rangle\rangle \mid a_1 = [a_{1,1}, \ldots, a_{1,\pi}], a_2 = [a_{2,1}, \ldots, a_{2,\rho}], a_{i,j}, b \in \mathbb{N}\}\\
&\delta_{\pi\rho} = \{\langle a_1 +\!\!+ \ldots +\!\!+ a_\pi, [a_1, \ldots, a_n]\rangle \mid a_i = [a_{i,1}, \ldots, a_{i,\rho}], a_{i,j} \in \mathbb{N}\}\\
&F_\rho = \{\langle f, \langle a, [b_1, \ldots, b_\rho]\rangle\rangle | \forall b_i. \exists c. c \in a \wedge \langle c, b_i\rangle \in f,\ a, f \in \mathbb{N}\}.
\end{aligned}
$$

This R-LCA also supports Booleans by defining $T = \{1\}$, $F = \{0\}$.

As a final prerequisite towards constructing a realisability quantitative CwF, we recall the definition of the category of assemblies [24, §2.3].

▶ **Definition 6.** *Let $\mathcal{A}$ be an R-LCA. An* assembly *is a pair $\Gamma = (|\Gamma|, \vDash_\Gamma)$, where $|\Gamma|$ is a set, and $\vDash_\Gamma$ is a relation $\vDash_\Gamma \subseteq \mathcal{A} \times |\Gamma|$. A morphism of assemblies from Γ to Δ is a function $f : |\Gamma| \to |\Delta|$, which is* realisable *— there exists $a_f \in \mathcal{A}$, such that $a \vDash_\Gamma \gamma$ implies $a_f \cdot a \vDash_\Delta f(\gamma)$.*

We can think of $|\Gamma|$ as a set of extensional meanings, and $a \vDash_\Gamma x$ as saying that the "program" $a \in \mathcal{A}$ realises the meaning of x. Usually, it is required that every $\gamma \in \Gamma$ has a realiser, but Atkey (personal communication) found the need to drop this condition due to technical reasons in the interpretation of dependent function types. This modified notion of assemblies and realisable functions still forms a category, using the B and I combinators to realise composition and identities. We now have all the pieces needed to describe Atkey's linear realisability model of QTT.

▶ **Proposition 7.** *Let $\mathcal{A}$ be the $\{0, 1, \omega^{<}\}$-LCA described in Example 5. There is a quantitative CwF where $\mathcal{C} = \mathcal{S}et$ and $\mathcal{L} = \mathcal{A}sm(\mathcal{A})$ the category of (modified) assemblies over $\mathcal{A}$, with $U(\Gamma) = |\Gamma|$. Types over Δ are given by Δ-indexed families of assemblies. Unresourced terms are given by set-theoretic functions, whereas resourced terms are realisable functions. Scaling of assemblies is defined by $\rho(\Gamma) = (|\Gamma|, \vDash_{\rho\Gamma})$ where $a \vDash_{\rho\Gamma} \gamma$ if there exists $b \in \mathcal{A}$, such that $a = \,!_\rho b$ and $b \vDash_\Gamma \gamma$. Similarly context addition is defined by the realisability relation where $a \vDash_{\Gamma_1+\Gamma_2} \gamma$ if there exist $b, c \in \mathcal{A}$, such that $a = [b, c]$ with $b \vDash_{\Gamma_1} \gamma$ and $c \vDash_{\Gamma_2} \gamma$.*

Proof. Atkey [6, §4.2] shows that the given data forms a quantitative category with families interpreting dependent tensor types, Booleans, and dependent function types, where the types are interpreted as follows: for dependent tensor types, we have $|(x \overset{\rho}{:} S) \otimes T| = \{(s, t) \mid s \in |S|, t \in |T(s)|\}$, with $a \vDash_{(x \overset{\rho}{:} S) \otimes T} (s, t)$ if there exists b, c such that $a = [!_\rho b, c]$, $b \vDash_S s$, and $c \vDash_{T(s)} t$. Booleans are interpreted using that they are supported in $\mathcal{A}$. Finally dependent function types are interpreted by $|(x \overset{\rho}{:} S) \to T| = \Pi_{x:|S|}|T(s)|$ where $a \vDash_{(x \overset{\rho}{:} S) \to T} f$ if $b \vDash_S s$ implies $a \cdot !_\rho b \vDash_{T(s)} f(s)$.[1] We briefly sketch the interpretation for the remaining type formers we have introduced: We have $|\mathbf{I}| = |\top| = \{\star\}$ and $|\mathbf{0}| = \emptyset$, with $I \vDash_{\mathbf{I}} \star$ and $x \vDash_\top \star$ for any $x \in \mathcal{A}$. Further we have $|(x : S) \,\&\, T| = \{(s, t) \mid s \in |S|, t \in |T(s)|\}$ and $|S \oplus T| = |S| + |T|$, with $\mathsf{case}(x, y) \vDash_{(x:S)\,\&\,T} (s, t)$ if $x \vDash_S s$ and $y \vDash_{T(s)} t$, and $[T, x] \vDash_{S \oplus T} \mathsf{inl}\, s$ if $x \vDash_S s$, and similarly $[F, y] \vDash_{S \oplus T} \mathsf{inr}\, t$ if $y \vDash_T t$ — for $S \,\&\, T$, the realisers for the projections can choose which realiser of x and y they want, and dually for $S \oplus T$, the realisers for the injections tag themselves with a Boolean T or F so that the realiser for the eliminator knows which case it is in. Finally the extensional identity type is interpreted as a subsingleton $|\mathsf{Id}(s, t)| = \{\star \mid [\![s]\!] = [\![t]\!]\}$, with a trivial realiser $I \vDash_{\mathsf{Id}(s,t)} \star$. ◀

We will use the above model to show that our proposed induction principles for data types are sound, by interpreting them in the model.

3 Data Types in Ordinary Type Theory

In this section, we recall how data types are usually presented in ordinary type theory.

3.1 Initial Algebra Semantics

An F-algebra for an endofunctor $F : \mathcal{C} \to \mathcal{C}$ is a pair (A, a), where A is an object of $\mathcal{C}$ and $a : F(A) \to A$ is a $\mathcal{C}$-morphism. A morphism between F-algebras (A, a) and (B, b) is a map $f : A \to B$ in $\mathcal{C}$, such that $f \circ a = b \circ F(f)$, i.e., the following diagram commutes:

$$\begin{tikzcd}
F(A) \arrow[r, "a"] \arrow[d, "F(f)"] & A \arrow[d, "f"] \\
F(B) \arrow[r, "b"] & B
\end{tikzcd}$$

Identity morphisms in $\mathcal{C}$ are F-algebra morphisms, and F-algebra morphisms compose. Hence F-algebras and their morphisms form a category. An initial F-algebra is an initial object

[1] Unfortunately the proof in Atkey [6, §4.2] overlooked the need to also allow potentially unrealised elements in $|(x \overset{\rho}{:} S) \to T|$, which is needed to achieve the curry-uncurry isomorphism for unresourced terms in Atkey [6, Def. 3.5].

in this category, that is, it is an F-algebra (A, a) with a unique morphism fold_B to every F-algebra B. We often write μF or $\mu X.F(X)$ for the initial F-algebra.

Initial F-algebras model inductively defined data types: the algebra map $a : F(A) \to A$ represents the constructors of the data type, and the unique morphism $\mathsf{fold}_B : A \to B$ gives an "iteration principle" for defining functions out of the data type, à la pattern matching definitions — as we will see later, this "non-dependent" elimination rule together with uniqueness is actually equivalent to full dependent elimination for many functors F. This equivalence makes crucial use of the extensional identity type.

Concretely, working in the category of types and functions, we can build up many data types as initial algebras of *polynomial functors*, inductively generated by the following grammar:

$$F, G ::= \mathsf{Id} \mid \mathsf{Const}_A \mid F \times G \mid F + G \mid \;\mid A \to F \tag{1}$$

where Id is the identity functor, $\mathsf{Const}_A(X) = A$, and $\times$, $+$, $A \to F$ are defined pointwise. For example, the data type of natural numbers is the initial algebra of $\mathsf{Const}_\mathbf{1} + \mathsf{Id}$, and the data type of binary trees with elements of A stored at the leaves is the initial algebra of the functor $A + \mathsf{Id} \times \mathsf{Id}$. Note that all types generated by (1) are strictly positive, in the sense that no input variable occurs to the left of an arrow. This is important for initial algebras to exist.

3.2 Containers

Containers are based on a shapes-and-positions metaphor for data types, meant to represent how concrete data are stored at locations in memory. For example, every list $\ell : \mathsf{List}\, X$ can be uniquely represented by a natural number $n : \mathbb{N}$ (given by the length of the list), together with a function $f : \mathsf{Fin}\, n \to X$, which returns the elements of the list at the given position. The number $n : \mathbb{N}$ represents the *shape* of the list, and $\mathsf{Fin}\, n$ is the type of *positions* of the given shape. In general, we have:

▶ **Definition 8** (Abbott, Altenkirch and Ghani [1]). *A* container $S \lhd P$ *is given by a type* $S : \mathsf{Type}$ *of shapes, and a type family* $P : S \to \mathsf{Type}$ *of positions. Its* extension *is given by the operation* $[\![S \lhd P]\!] : \mathsf{Type} \to \mathsf{Type}$ *defined by*

$$[\![S \lhd P]\!]\, X = (s : S) \times (P(s) \to X) \ .$$

More generally, containers can be presented in arbitrary locally Cartesian closed categories, but we restrict ourselves to the category of types here. The extension of a container formalises the intuition that an instantiation of the container is given by choosing a shape, and then a a payload for each position. This action is functorial.

▶ **Proposition 9.** *Let* $S \lhd P$ *be a container. The extension* $[\![S \lhd P]\!]$ *extends to a functor* $\mathsf{Type} \to \mathsf{Type}$*, defined by* $[\![S \lhd P]\!](g)\,(s, f) = (s, g \circ f)$. ◀

The action on morphisms evidently preserves identities and composition. We call any functor isomorphic to one of the form $[\![S \lhd P]\!]$ a container functor. Container functors are closed under many type formers and operations, for example:

$$\begin{aligned}
\mathsf{Id} &\cong [\![\mathbf{1} \lhd (\lambda\_.\mathbf{1})]\!] \\
\mathsf{Const}_A &\cong [\![A \lhd (\lambda\_.\mathbf{0})]\!] \\
[\![S \lhd P]\!] \times [\![S' \lhd P']\!] &\cong [\![\big(S \times S'\big) \lhd \big(\lambda(s, s').P(s) + P'(s')\big)]\!] \\
[\![S \lhd P]\!] + [\![S' \lhd P']\!] &\cong [\![\big(S + S'\big) \lhd [P, P']\,]\!] \\
A \to [\![S \lhd P]\!] &\cong [\![\big(A \to S\big) \lhd \big(\lambda f.(a : A) \times P(f(a))\big)]\!]
\end{aligned} \tag{2}$$

Using the above isomorphisms, one can show that any polynomial functor in the sense of (1) can be represented as a container.

▶ **Theorem 10** (Dybjer [15]). *In extensional type theory, for any polynomial functor F, there is a container $S_F \lhd P_F$ such that $F \cong [\![S_F \lhd P_F]\!]$.* ◀

Hence in extensional type theory, the problem of constructing initial algebras for polynomial functors reduces to the problem of constructing initial algebras of containers, which are given by Martin Löf's W-type[2]. As a consequence, we can restrict our attention to containers, that are easier to work with compared to polynomial functors, since they are not inductively generated.

4 Quantitative Containers

Based on their success for data types in ordinary type theory, it seems reasonable to try to generalise containers to the quantitative setting. Since categorical models of QTT are monoidal closed rather than Cartesian closed, they are in particular not locally Cartesian closed, and so, this is not a question of interpreting containers directly.

4.1 Quantitative Container Functors on the Category of Closed Types and Linear Functions

Working internally in QTT, we can keep the same notion of a container $S \lhd P$ as given by $S : \mathsf{Type}$ and $P : S \to \mathsf{Type}$. Since types are checked in the 0-fragment, we do not need to worry about linear uses of $s : S$ when forming $P(s) : \mathsf{Type}$. However from now on, we change the extension of the container from using dependent pairs and functions, to using dependent tensors and linear functions respectively:

▶ **Definition 11.** *A* quantitative container $S \lhd P$ *is given by a QTT type* $S : \mathsf{Type}$ *of shapes, and a QTT type family* $P : S \to \mathsf{Type}$ *of positions. Its* extension *is given by the operation* $[\![S \lhd P]\!] : \mathsf{Type} \to \mathsf{Type}$ *defined by*

$$[\![S \lhd P]\!]\, X = (s \overset{1}{:} S) \otimes (P(s) \overset{1}{\to} X) \enspace .$$

As expected, container extensions are still functorial, if we move to the category $\mathsf{Type}_{\mathrm{Lin}}$ of closed types and *linear* functions: a morphism from a type X to a type Y is given by a function $\vdash f : X \overset{1}{\to} Y$.

▶ **Proposition 12.** *Let $S \lhd P$ be a container. The extension $[\![S \lhd P]\!]$ extends to a functor $\mathsf{Type}_{Lin} \to \mathsf{Type}_{Lin}$, defined by $[\![S \lhd P]\!](g)\, x = \ let\ (s, f) = x\ in\ (s, g \circ f)$.* ◀

The above proposition can be generalised to a category of types and functions over an arbitrary, fixed context of the shape $\Gamma = 0\Gamma$, i.e. a context where all variables are annotated with 0.

▶ **Example 13.** We can define the list container $\mathbb{N} \lhd \mathsf{Fin}$ as before, but note that its use is rather complicated, due to linearity constraints: to define a function out of $[\![\mathbb{N} \lhd \mathsf{Fin}]\!]\, X$, we have to make use of both the length n and the payload function $f : \mathsf{Fin}(n) \to X$ exactly once. As an example, consider a slightly unusual Maybe container, given by $M =$

[2] Recently, Hugunin [25] showed how this result can be extended also to intensional type theory.

$\mathbf{2} \lhd \big(\lambda b.\ \text{if } b \text{ then } \mathbf{I} \text{ else } \mathsf{Fin}\,0\big)$: we have two shapes, one with trivial positions (representing a just value), and one with $\mathsf{Fin}\,0$ positions (representing $\mathsf{nothing}$). Now we can write a function $\mathsf{head} : [\![\mathbb{N} \lhd \mathsf{Fin}]\!]\, X \xrightarrow{1} [\![M]\!]\, X$, which given a tensor tuple (n, f) first inspects the length n; if it is non-zero, $f : \mathsf{Fin}\,(n' + 1) \xrightarrow{1} X$ and we can return $\mathsf{just}\,(f\,0)$, otherwise if $n = 0$ we can return $f : \mathsf{Fin}\,0 \xrightarrow{1} X$ itself as the payload function in the $\mathsf{nothing}$ case, thus using both the length and the payload of the list. However many other plausibly linear operations on lists can not be written with this representation — as we will see, this hints at a deficiency of the container representation of data types in a quantitative setting.

▶ **Example 14.** Hancock and Hyvernat [21] suggest to think of a container $Q \lhd A$ as an interaction structure: the shapes Q represent questions, or commands, and the positions A represent answers, or responses. An element of $[\![Q \lhd A]\!]\, X$ is thus a tuple (q, f) where q is a question, and $f : A(q) \xrightarrow{1} X$ is a linear function ready to give an element of X for every answer to q. An initial algebra $D = \mu X.[\![Q \lhd A]\!]\, X$ of $[\![Q \lhd A]\!]$ thus intuitively consists of wellfounded "dialogue trees" (q, f) where $q : Q$ is a question, and $f : A(q) \xrightarrow{1} D$ is a function ready to supply a dialogue subtree for each possible answer to the question. Linearity here means that to consume such a dialogue tree, we firstly need to consume the question, and then we get to choose *exactly one* answer to the question to continue the dialogue, and so on, until we reach a question with no answers. Similar to when modelling stateful operations [33], linearity prevents us from "going back in time" and speculatively trying a different answer. Dually, when constructing such a dialogue tree, the typing rules allow us to use the same resources when constructing different subtrees — only one subtree is eventually going to be explored anyway.

4.2 Closure Under Type Formers

Unfortunately, many of the isomorphisms in (2) do not carry over from ordinary containers to quantitative containers, due to the bifurcation into additive and multiplicative type formers in the linear setting.

▶ **Theorem 15.** *The identity functor is a quantitative container, but constant functors are not, in general.*

Proof. The identity functor is represented by the quantitative container $\mathbf{I} \lhd (\lambda\_.\mathbf{I})$, since $\mathbf{I} \otimes A \cong A$ and $\mathbf{I} \xrightarrow{1} A \cong A$ by Lemma 3. For showing that quantitative containers are not closed under constant functors, note that every model of ordinary type theory is also a model of QTT, with vacuous resource tracking [6, Prop 3.3]. Hence if $\mathsf{Const}_A \cong [\![S \lhd P]\!]$, the same isomorphism must hold if all linear type formers are replaced by their Cartesian variants in the definition of S and P, because that would be the interpretation of the isomorphism in a vacuous model. Further, since the extension functor $[\![-]\!]$ is full and faithful for ordinary containers [1, Thm. 3.4], if $\mathsf{Const}_A \cong [\![S \lhd P]\!]$ in a vacuous model, then in that model we must have $S \cong A$ and $(P(s) \to X) \cong \mathbf{1}$ by (2). Going back to the linear world, we must hence for each $s : S$ and type X have $(P(s) \xrightarrow{1} X) \cong \mathbf{I}$. In particular, for $X = \mathbf{0}$, there is a function of type $P(s) \xrightarrow{1} \mathbf{0}$, which implies $P(s) \cong \mathbf{0}$ since $\mathbf{0}$ is initial. But this absurd: since $(\mathbf{0} \xrightarrow{1} X) \cong \top$, we then would have $\top \cong \mathbf{I}$, but this isomorphism does not hold in all models. Hence there cannot be such a quantitative container $S \lhd P$. ◀

It is not hard to see that quantitative containers are closed under $\oplus$, as the isomorphism for $+$ in (2) is already linear. However the isomorphisms for closure under all the other type formers are not, and so quantitative containers are not closed under function spaces, $\otimes$, or &. This is quite a blow for their usefulness as a general framework for data types — as a consequence, we will consider alternatives in Section 5.

4.3 Elimination Rules and Induction Principles

As we have seen in Example 14, it is sometimes helpful to consider initial algebras W of quantitative container functors $[\![S \triangleleft P]\!]$. The algebra map $c : [\![S \triangleleft P]\!](W) \xrightarrow{1} W$ corresponds to the introduction rule of the type, and the mediating morphism into any other $[\![S \triangleleft P]\!]$-algebra $B : \mathsf{Type}$ corresponds to an elimination rule:

$$\frac{\vdash b : \big((s \overset{1}{:} S) \otimes (P(s) \xrightarrow{1} B)\big) \xrightarrow{1} B}{\vdash \mathsf{fold}_{(B,b)} : W \xrightarrow{1} B}$$

with computation rule $\mathsf{fold}_{(B,b)}(c(x)) = b([\![S \triangleleft P]\!](\mathsf{fold}_{(B,b)})\, x)$ given by the fact that $\mathsf{fold}_{(B,b)}$ is an algebra morphism. A priori, this only gives a *non-dependent* elimination rule (also known as a recursion principle), but by exploiting the uniqueness of the mediating map, we can also derive the following *dependent* elimination rule (induction principle) for any $\vdash Q : W \to \mathsf{Type}$:

$$\frac{\vdash m : (s \overset{1}{:} S) \to (h \overset{0}{:} P(s) \xrightarrow{1} W) \to ((p \overset{1}{:} P(s)) \to Q[h(p)]) \xrightarrow{1} Q[c(s,h)]}{\vdash \mathsf{elim}(Q,m) : (x \overset{1}{:} W) \to Q[x]} \qquad (3)$$

This rule says that to prove $Q[x]$ for every $x : W$ using one copy of x, it is enough to prove $Q[c(s,h)]$ for $s : S$ and $h : P(s) \xrightarrow{1} W$, assuming $Q[h(p)]$ already holds for every $p : P(s)$, using one copy of s and the induction hypothesis, but zero copies of h. A priori, it is perhaps not completely obvious that this is the right usage annotation of the variables involved. Indeed, a noteworthy feature of Theorem 16 below is that these annotations naturally fall out from its proof, so that we can use it to derive the form of of the induction principle in a principled manner in the quantitative setting.

In fact, just like in ordinary type theory, initiality and induction are actually equivalent principles: An algebra is initial if and only if it supports induction. This result and construction seems to have been discovered several times by several authors, for example Dybjer and Setzer [16], Hermida and Jacobs [23], and Awodey, Gambino and Sojakova [7]. Our contribution here is to adapt the proof to also account for linearity.

▶ **Theorem 16.** *Let $(W, c : [\![S \triangleleft P]\!](W) \xrightarrow{1} W)$ be an $[\![S \triangleleft P]\!]$-algebra. The algebra W is initial if and only if there is a term $\mathsf{elim}(Q,m)$ as in (3) for every $w \overset{0}{:} W \vdash Q : \mathsf{Type}$, satisfying the computation rule $\mathsf{elim}(Q, c(x)) =$ let $(s,h) = x$ in $m\, s\, h\, (\lambda p.\mathsf{elim}(Q, h(p)))$.*

Proof. Assuming that (W, c) is initial and the premises of the elimination rule, we build an $[\![S \triangleleft P]\!]$-algebra on the dependent tensor type $(w \overset{0}{:} W) \otimes Q$, and get a unique mediating morphism $\mathsf{fold} : W \xrightarrow{1} (w \overset{0}{:} W) \otimes Q$ by initiality. We compose with the second projection $\mathsf{snd} : (x \overset{1}{:} (w \overset{0}{:} W) \otimes Q) \to Q[\mathsf{fst}(x)]$ to get a map $(x \overset{1}{:} W) \to Q[\mathsf{fst}(\mathsf{fold}(x))]$. Note that the use of second projection is admissible due to the annotation of the first component $w \overset{0}{:} W$ — we are free to dispose of w, since it is used 0 times. To show that $\mathsf{snd} \circ \mathsf{fold}$ has the right type, we need to show that $Q[\mathsf{fst}(\mathsf{fold}(x))] = Q[x]$ for every $x : W$, but as this is a type equality, unrestricted use of terms is permissible. The map $\mathsf{fst} : (w \overset{0}{:} W) \otimes Q \xrightarrow{1} W$ is an

$[\![S \lhd P]\!]$-algebra morphism, and thus the composite $\mathsf{fst} \circ \mathsf{fold} : W \xrightarrow{1} W$ is also one:

$$\begin{tikzcd}
{[\![S \lhd P]\!](W)} \arrow[r, "c"] \arrow[d] & W \arrow[d, "\mathsf{fold}"] \\
{[\![S \lhd P]\!]((w \overset{0}{:} W) \otimes Q)} \arrow[r] \arrow[d] & (w \overset{0}{:} W) \otimes Q \arrow[d, "\mathsf{fst}"] \\
{[\![S \lhd P]\!](W)} \arrow[r, "c"] & W
\end{tikzcd}$$

Thus $\mathsf{fst} \circ \mathsf{fold} = \mathsf{id}$ holds by uniqueness of the mediating morphism out of W. The computation rule is exactly that fold is an $[\![S \lhd P]\!]$-algebra morphism.

For the converse direction, assume that the elimination rule holds for (W, c). Let $(A, f : [\![S \lhd P]\!](A) \to A)$ be an arbitrary $[\![S \lhd P]\!]$-algebra and define the type $Q[x] := A$ for all $x : W$. Hence the type of the method in the elimination simplifies to

$$m \overset{1}{:} (s \overset{1}{:} S) \to (h \overset{0}{:} P(s) \xrightarrow{1} W) \to ((p \overset{1}{:} P(s)) \to A) \xrightarrow{1} A$$

which we can define by $m = \lambda s.\lambda h.\lambda y.f(s, y)$, since we have 0 copies of h, and thus do not need to use it. The elimination principle gives a function $\mathsf{fold} := \mathsf{elim}(A, m) : W \xrightarrow{1} A$, which by the computation rule is an $[\![S \lhd P]\!]$-algebra morphism. We show uniqueness of fold as follows: given another $[\![S \lhd P]\!]$-algebra morphism $g : W \xrightarrow{1} A$, we use the elimination principle and equality reflection to prove $g(x) = \mathsf{fold}(x)$ for every $x : W$. For x of the form $c(s, f)$, this follows linearly from both g and fold being $[\![S \lhd P]\!]$-algebra morphisms and the induction hypothesis, and the result follows by equality reflection. ◀

5 Quantitative Polynomial Functors

As we have seen, quantitative containers give a seemingly well behaved notion of data types in quantitative type theory: they are functorial, and their algebras support induction principles if and only if they are initial, which is a property that is usually easier to verify in models. However, there is a caveat — most quantitative versions of standard data types are not quantitative containers in the above sense, because quantitative containers are not closed under many type formers, as discussed in Section 4.2.

Consider, for example, the natural numbers, the initial algebra of the polynomial functor $F(X) = \mathbf{1}+X$, or binary trees, the initial algebra of $G(X) = A+X\times X$. Their representations as initial algebras of containers in Theorem 10 crucially depend on the isomorphisms $(\mathbf{0} \to X) \cong \mathbf{1}$ and $(\mathbf{2} \to X) \cong X \times X$, respectively. If we consider the corresponding QTT data types given by $F(X) = \mathbf{I} \oplus X$ and $G(X) = A \oplus (X \otimes X)$, we see using Lemma 3 that the QTT counterparts of the above isomorphisms do not hold: $(\mathbf{0} \xrightarrow{1} X) \cong \top \ncong \mathbf{I}$ and $(\mathbf{2} \xrightarrow{1} X) \cong X \,\&\, X \ncong X \otimes X$.

5.1 A Grammar for Quantitative Polynomial Functors

Since we can no longer reduce all data types we are interested in to quantitative containers, we instead resort to generating them inductively. This follows the same pattern as for polynomial functors in the grammar (1), except that we now have multiplicative and additive versions of many type formers.

▶ **Definition 17.** *The class of* quantitative polynomial type expressions *is inductively generated by the following grammar:*

$$F, G ::= \mathsf{Id} \mid \mathsf{Const}_A \mid F \otimes G \mid F \oplus G \mid F \mathbin{\&} G \mid A \xrightarrow{1} F$$

If the type expression is generated without making use of the $A \xrightarrow{1} F$ production, we call it finitary*.*

A simple induction on how the type expression is generated proves that all quantitative polynomial type expressions are functors of type $\mathsf{Type}_{\mathrm{Lin}} \to \mathsf{Type}_{\mathrm{Lin}}$. Hence we are justified in calling them polynomial functors.

▶ **Example 18.** A functor describing the natural numbers is given by the finitary type expression $F = \mathsf{Const}_{\mathbf{I}} \oplus \mathsf{Id}$. An element of $F(X)$ is either $\mathsf{inl}\,\star$, representing 0, or of the form $\mathsf{inr}\,n$, representing the successor of n. A function $F(X) \xrightarrow{1} A$ uses no resources for the zero case, and gets one copy of n in the successor of n case. This is different from an attempted quantitative container representation of F, which would be given by $\mathbf{2} \triangleleft P$, where $P(\mathsf{true}) = \mathbf{0}$ and $P(\mathsf{false}) = \mathbf{I}$, since the representation of 0 there would also have a "junk" term of type $(\mathbf{0} \xrightarrow{1} X) \cong \top$ around.

▶ **Example 19.** Similarly, a finitary polynomial functor describing binary trees is given by $G = \mathsf{Const}_A \oplus (\mathsf{Id} \otimes \mathsf{Id})$. An element is either of the form $\mathsf{inl}\,a$ for some $a : A$, representing a leaf with label a, or of the form $\mathsf{inr}\,(\ell, r)$, representing a node with subtrees ℓ and r. A function $G(X) \xrightarrow{1} B$ must be able to deal with either getting one copy of $a : A$, or one copy each of subtrees ℓ and r. In contrast, an attempted quantitative container representation of G would be given by $(A \oplus \mathbf{I}) \triangleleft P$, where $P(\mathsf{inl}\,a) = \mathbf{0}$ and $P(\mathsf{inr}\,\star) = \mathbf{2}$. This would have the same "junk term" problem for leaves as zero does in Example 18, but in addition, a function $[\![(A \oplus \mathbf{I}) \triangleleft P]\!] \xrightarrow{1} B$ would only get access to *one* of the two subtrees ℓ and r. If this is the intended use case, like in Example 14, then one can instead change the quantitative polynomial functor to $G' = \mathsf{Const}_A \oplus (\mathsf{Id} \mathbin{\&} \mathsf{Id})$.

5.2 Elimination Rules and Induction Principles

In order to consider initial algebras, a functor is all we need, but to formulate induction principles, we now need to construct some additional machinery: we need to explain what the type of induction hypothesis is for a given functor, and what the computation rule should be. For quantitative containers, we could do this directly, but since quantitative polynomial functors are inductively generated, we also need to proceed inductively. Our main tool is the predicate lifting [23] of polynomial functors, which will be used as the type of induction hypothesis.

▶ **Definition 20.** *Let F be a quantitative polynomial functor. We define its predicate lifting*

$\widehat{F} : (Q : X \to \mathsf{Type}) \to (F(X) \to \mathsf{Type})$ *by induction on* F*:*

$$\begin{aligned}
\widehat{\mathsf{Id}}(Q, x) &= Q(x)\\
\widehat{\mathsf{Const}_A}(Q, x) &= (a \overset{1}{:} A) \otimes (a = x)\\
\widehat{F \otimes G}(Q, x) &= \widehat{F}(Q, \mathsf{proj}_1\, x) \otimes \widehat{G}(Q, \mathsf{proj}_2\, x)\\
\widehat{F \oplus G}(Q, x) &= ((x_1 \overset{0}{:} F(X)) \otimes (p \overset{1}{:} x = \mathsf{inl}\, x_1) \otimes \widehat{F}(Q, x_1)) \oplus\\
&\qquad ((x_2 \overset{0}{:} G(X)) \otimes (p \overset{1}{:} x = \mathsf{inr}\, x_2) \otimes \widehat{G}(Q, x_2))\\
\widehat{F \,\&\, G}(Q, x) &= \widehat{F}(Q, \mathsf{proj}_1\, x) \,\&\, \widehat{G}(Q, \mathsf{proj}_2\, x)\\
\widehat{A \xrightarrow{1} F}(Q, x) &= (a \overset{1}{:} A) \to \widehat{F}(Q, x(a)).
\end{aligned}$$

The lifting follows the same structure as the underlying functor. Note that we can use projections $\mathsf{proj}_i := \lambda x.\ \mathrm{let}\ (x_1, x_2) = x\ \mathrm{in}\ x_i$ in the $F \otimes G$ case, as we are defining a type, and hence we are in the 0 fragment. The only possibly surprising case is the lifting of constant functors $\widehat{\mathsf{Const}_A}(Q, x) = (a \overset{1}{:} A) \otimes (a = x)$; in a non-quantitative setting, this would be a complicated way to define $\widehat{\mathsf{Const}_A}(Q, x) = \mathbf{1}$, since, in the language of homotopy type theory, singletons $(a : A) \times (a = x)$ are contractible [36, Lem. 3.11.8]. However the point is that we get one "extra" copy a of x which can be used even when we have zero copies of x available. This is crucial for the proof of Lemma 24 below.

We will use predicate liftings to formulate the notion of induction hypothesis for F-algebras for a quantitative polynomial functor F. For formulating computation rules, we will furthermore make use of the following lemma, which states that each predicate lifting has a "functorial action" on *dependent* functions:

▶ **Lemma 21.** *Let F be a quantitative polynomial functor. If $f : (x \overset{1}{:} X) \to Q(x)$ then we can define $\widehat{F}(f) : (y \overset{1}{:} F(X)) \to \widehat{F}(Q, y)$. In addition, if $g : Y \xrightarrow{1} X$, then $\widehat{F}(Q, F(g)(y)) = \widehat{F}(Q \circ g, y)$, and $\widehat{F}(f \circ g) = \widehat{F}(f) \circ F(g)$.* ◀

The proof is again a simple induction on the buildup of F. Since there are no identity dependent functions or compositions of dependent functions in general for this fixed form, it does not make sense to ask for $\widehat{F}$ to preserve neither identities or composition in general, beyond the "mixed" $\widehat{F}$–F preservation stated in the lemma. However Lemma 25 below implies that $\widehat{F}$ does preserve identities and compositions when these make sense.

We now have all the ingredients we need to define what it means for an algebra to support elimination and computation rules. Let F be a quantitative polynomial functor, and (W, c) an F-algebra. The dependent elimination rule (or induction principle) for W, for any predicate $\vdash Q : W \to \mathsf{Type}$, is stated as follows:

$$\frac{\vdash m : (y \overset{0}{:} F(W)) \to \widehat{F}(Q, y) \xrightarrow{1} Q(c(y))}{\vdash \mathsf{elim}(Q, m) : (x \overset{1}{:} W) \to Q(x)} \tag{4}$$

with computation rule $\mathsf{elim}(Q, m)(c(y)) = m\, y\, (\widehat{F}(\mathsf{elim}(Q, m), y))$. Note how the computation rule is making use of the action of $\widehat{F}$ on dependent functions from Lemma 21.

▶ **Example 22.** Recall from Example 18 that the quantitative polynomial functor $F = \mathsf{Const}_{\mathbf{I}} \oplus \mathsf{Id}$ describes the data type of natural numbers. The type of induction hypothesis for $\mathsf{zero} := c(\mathsf{inl}\, \star)$ is, up to isomorphism, $\widehat{F}(Q, \mathsf{inl}\, \star) \cong \mathbf{I}$, i.e., it contains no information, and the induction hypothesis for $\mathsf{suc}\,n := c(\mathsf{inr}\,n)$ is isomorphic to $\widehat{F}(Q, \mathsf{inr}\,n) \cong \widehat{\mathsf{Id}}(Q, n) = Q(n)$, as expected. We would thus expect the induction principle to be familiarly stated as follows

$$\frac{\vdash m_z : Q(\mathsf{zero}) \quad \vdash m_s : (n \overset{0}{:} \mathbb{N}) \to Q(n) \xrightarrow{1} Q(\mathsf{suc}\,n)}{\vdash \mathsf{elim}(Q, m) : (x \overset{1}{:} \mathbb{N}) \to Q(x)}$$

and indeed it can be, up to isomorphism. An important point is that when translating between $m : (y \overset{0}{:} F(W)) \to \widehat{F}(Q, y) \xrightarrow{1} Q(c(y))$ and m_z and m_s as above, we cannot make case distinctions on $y \overset{0}{:} F(W)$, since we have zero copies of y available. Instead, we have to split on the $\widehat{F}(Q, y)$ argument, which in turn will refine y. This is why the definition of $\widehat{F \oplus G}(Q, x)$ is designed the way it is, rather than just giving cases for $\widehat{F \oplus G}(Q, \mathsf{inl}\,x_1)$ and $\widehat{F \oplus G}(Q, \mathsf{inr}\,x_2)$ directly.

▶ **Example 23.** The quantitative polynomial functor $G = \mathsf{Const}_A \oplus (\mathsf{Id} \otimes \mathsf{Id})$ from Example 19 describes binary trees. For leaves, we have $\widehat{G}(Q, \mathsf{inl}\,x) \cong (a \overset{1}{:} A) \otimes (a = x)$, and for nodes we have $\widehat{G}(Q, \mathsf{inr}\,x) \cong Q(\mathsf{proj}_1\,x) \otimes Q(\mathsf{proj}_2\,x)$. Hence the induction principle becomes

$$\frac{\vdash m_l : (a \overset{1}{:} A) \to Q(\mathsf{leaf}\,a) \quad \vdash m_n : (\ell \overset{0}{:} \mathsf{Tree}_A) \to (r \overset{0}{:} \mathsf{Tree}_A) \to Q(\ell) \xrightarrow{1} Q(r) \xrightarrow{1} Q(\mathsf{node}\,\ell\,r)}{\vdash \mathsf{elim}(Q, m) : (t \overset{1}{:} \mathsf{Tree}_A) \to Q(t)}$$

Note how the method m_l gets one rather than zero copies of $a : A$, thanks to the definition of $\widehat{G}(Q, \mathsf{inl}\,x) \cong (a \overset{1}{:} A) \otimes (a = x)$.

We now aim to show that initiality and induction are equivalent also for quantitative polynomial functors. The proof follows the same pattern as Theorem 16, but we need some additional lemmas. Firstly, we need that F distributes over dependent tensor products of the form $(w \overset{0}{:} W) \otimes Q$ in an appropriate sense:

▶ **Lemma 24.** *For a quantitative polynomial functor F, we have*

$$\mathsf{dist}_F : F((x \overset{0}{:} W) \otimes Q(x)) \xrightarrow{1} (y \overset{0}{:} F(W)) \otimes \widehat{F}(Q, y)$$

with $\mathsf{proj}_1 \circ \mathsf{dist}_F = F(\mathsf{proj}_1)$ as an equation in the 0-fragment. ◀

Secondly, for deriving initiality from induction, we need that $\widehat{F}$ basically reduces to F for constant predicates:

▶ **Lemma 25.** *If $Q(x) = A$ for every $x : X$, i.e., Q is constant, then $\widehat{F}(Q, y) \cong F(A)$ for every $y : F(X)$, and $\widehat{F}(f) \cong F(f)$ for every $f : X \xrightarrow{1} A$.* ◀

Armed with these lemmas, we can now attack our main theorem:

▶ **Theorem 26.** *Let F be a quantitative polynomial functor and $(W, c : F(W) \xrightarrow{1} W)$ an F-algebra. The algebra W is initial if and only if there is a term $\mathsf{elim}(Q, m)$ as in (4) for every $w \overset{0}{:} W \vdash Q : \mathsf{Type}$, satisfying the computation rule $\mathsf{elim}(Q, m)(c(y)) = m\,y\,(\widehat{F}(\mathsf{elim}(Q, m), y))$.*

Proof. Assuming that (W, c) is initial, and given $m : (y \overset{0}{:} F(W)) \to \widehat{F}(Q, y) \xrightarrow{1} Q(c(y))$, we construct an F-algebra

$$F((x \overset{0}{:} W) \otimes Q(x)) \xrightarrow{\mathsf{dist}_F} (y \overset{0}{:} F(W)) \otimes \widehat{F}(Q, y) \xrightarrow{\langle c, m \rangle} (x \overset{0}{:} W) \otimes Q(x)$$

and by initiality, we get $\mathsf{fold}_{\langle c,m\rangle\circ\mathsf{dist}_F} : W \xrightarrow{1} (x \overset{0}{:} W) \otimes Q(x)$. By Lemma 24, the following diagram commutes, meaning that proj_1 is an algebra morphism from $(x \overset{0}{:} W) \otimes Q(x)$ to W in the 0-fragment:

$$\begin{tikzcd}
F((x \overset{0}{:} W) \otimes Q(x)) \arrow[r, "\mathsf{dist}_F"] \arrow[d, "F(\mathsf{proj}_1)"] & (y \overset{0}{:} F(W)) \otimes \widehat{F}(Q,y) \arrow[r, "\langle c,m\rangle"] \arrow[d, "\mathsf{proj}_1"] & (x \overset{0}{:} W) \otimes Q(x) \arrow[d, "\mathsf{proj}_1"] \\
F(W) \arrow[r, equal] & F(W) \arrow[r, "c"] & W
\end{tikzcd}$$

Hence by uniqueness, we have $\mathsf{proj}_1 \circ \mathsf{fold}_{\langle c,m\rangle\circ\mathsf{dist}_F} = \mathsf{id}$, and hence we can define $\mathsf{elim}(Q,m) = \mathsf{proj}_2 \circ \mathsf{fold}_{\langle c,m\rangle\circ\mathsf{dist}_F} : (y \overset{1}{:} W) \to Q(y)$. The computation rule follows from the fact that fold is an F-algebra morphism, and the second part of Lemma 21. Conversely, using Lemma 25 we can use the induction principle on a constant predicate to get a morphism $W \xrightarrow{1} B$ for an F-algebra B. We prove uniqueness with another instantiation of the induction principle, together with function extensionality. ◀

5.3 Initial Algebras of Finitary Quantitative Polynomial Functors in the Realisability Model

Initial algebras of a polynomial functor $F : \mathcal{C} \to \mathcal{C}$ need not necessarily exist for an arbitrary category $\mathcal{C}$. When the category in question is $\mathcal{S}et$, however, the initial algebra μF always exists for polynomial functors. We show that a similar result holds for finitary quantitative polynomial functors in the quantitative CwF from Proposition 7. We refer to that category as the realisability model $\mathcal{M}_r$.

Semantic types in $\mathcal{M}_r$ are interpreted by a collection of assemblies. Thus to construct the initial algebra of a quantitative polynomial functor F, we first need to define a type μF, endow it with a realisable structure map $c : F(\mu F) \to \mu F$, and finally show that the mediating morphism fold to any other F-algebra $(X, \alpha : F(X) \to X)$ is realisable. For simplicity of notation, we present the construction as in the empty context.

Let $\tilde{F} : \mathcal{S}et \to \mathcal{S}et$ designate the set functor obtained from F by replacing each linear connective with a Cartesian one. This is a polynomial functor, and hence has an initial algebra $(\mu\tilde{F}, \tilde{c} : \tilde{F}(\mu\tilde{F}) \to \mu\tilde{F})$. Our plan is to augment the $\tilde{F}$-initial algebra with realisability information following the structure of F. Denote by $\overline{\mathbf{n}}$ the encoding of the numeral n using tupling and Booleans in the underlying R-LCA $\mathcal{A} = \mathcal{P}(\mathbb{N})$ from Example 5.

▶ **Construction 27.** Let F be a quantitative polynomial functor and let $(\mu\tilde{F}, \tilde{c} : \tilde{F}(\mu\tilde{F}) \to \mu\tilde{F})$ be the initial algebra of $\tilde{F}$ in $\mathcal{S}et$. We define $(\Vdash_{\mu F} x) \subseteq \mathcal{A}$ by induction on $x \in \mu\tilde{F}$ and the buildup of F: it is sufficient to define $a \Vdash_{\mu F} \tilde{c}(y)$ for some $y \in \tilde{F}(\mu\tilde{F})$, assuming we have already defined the relation $a' \Vdash_{\mu F} z$ for all structurally smaller z.

- if $F = \mathsf{Id}$, then $y \in |\mu F|$ and we have already defined $(\Vdash_{\mu F} x) \subseteq \mathcal{A}$. We define $a \Vdash_{\mu F} \tilde{c}(y) \iff \exists b \in \mathcal{A},$ such that $b \Vdash_{\mu F} y \wedge a = [\overline{\mathbf{1}}, b]$.
- if $F = \mathsf{Const}_A$, then $y \in |A|$ and we define $a \Vdash_{\mu F} \tilde{c}(y) \iff \exists b \in \mathcal{A},$ such that $b \Vdash_A y \wedge a = [\overline{\mathbf{2}}, b]$.
- if $F = F' \otimes G'$, then there are some $y_1 \in |F'(\mu F)|$ and $y_2 \in |G'(\mu F)|$, such that $y = (y_1, y_2)$. Let $a \Vdash_{\mu F} \tilde{c}(y) \iff \exists b_1, b_2 \in \mathcal{A},$ such that $b_1 \Vdash_{F'(\mu F)} y_1 \wedge b_2 \Vdash_{G'(\mu F)} y_2 \wedge a = [\overline{\mathbf{3}}, [b_1, b_2]]$
- if $F = F' \oplus G'$, there are $y_1 \in |F'(\mu F)|$ and $y_2 \in |G'(\mu F)|$, such that $y = \mathsf{inl}(y_1)$ or $y = \mathsf{inr}(y_2)$.
  If $y = \mathsf{inl}(y_1)$, let $a \Vdash_{\mu F} \tilde{c}(y) \iff \exists b_1 \in \mathcal{A},$ such that $b_1 \Vdash_{F'(\mu F)} y_1 \wedge a = [\overline{\mathbf{4}}, [T, b_1]]$;
  if $y = \mathsf{inr}(y_2)$, let $a \Vdash_{\mu F} \tilde{c}(y) \iff \exists b_2 \in \mathcal{A},$ such that $b_2 \Vdash_{G'(\mu F)} y_2 \wedge a = [\overline{\mathbf{4}}, [F, b_2]]$.

- if $F = F' \,\&\, G'$, there are $y_1 \in |F'(\mu F)|$ and $y_2 \in |G'(\mu F)|$, such that $y = (y_1, y_2)$. Let $a \vDash_{\mu F} \tilde{c}(y) \iff \exists b_1, b_2 \in \mathcal{A}$, such that $b_1 \vDash_{F'(\mu F)} y_1 \wedge b_2 \vDash_{G'(\mu F)} y_2 \wedge a = [\mathbf{5}, \mathsf{case}(b_1, b_2)]$.
- if $F = A \xrightarrow{1} F'$, then $y \in |A \to F(\mu F)|$. Let $a \vDash_{\mu F} \tilde{c}(y) \iff \exists b \in \mathcal{A}$, s.t. $(\forall i \in A\ \forall a_i \in \mathcal{A}, a_i \vDash_A i \implies b \cdot a_i \vDash_{F'(\mu F)} y(i)) \wedge a = [\mathbf{6}, b]$.

The only perhaps slightly odd case is for the function space $F = A \xrightarrow{1} F'$, which is obtained through unwinding the definition of a realisable function. The following lemma is easily provable by induction on the buildup of F, using combinatory completeness to define the realisers.

▶ **Lemma 28.** *Let F be a quantitative polynomial functor and μF be constructed as in Construction 27. Then the function $c : F(\mu F) \to \mu F$ is realisable.* ◀

Similarly, we can prove that the mediating morphism fold is realisable by induction on its argument $x \in \mu\tilde{F}$, when F is finitary, i.e., when the rule $A \xrightarrow{1} F'$ is omitted. This is making essential use of the fact that fold is an $\tilde{F}$-algebra morphism in $\mathcal{S}et$.

▶ **Lemma 29.** *Let F be a finitary quantitative polynomial functor, and $(X, \alpha : F(X) \to X)$ be an F-algebra. Then the map $\mathsf{fold} : \mu F \to X$ is realisable.* ◀

It is important to understand why we had to assume that F was finitary: given a function $y \in |A \xrightarrow{1} F'(\mu F)|$, s.t. $x = c(y)$, we get a family of realisers $\{a_i | a_i \vDash_{F'(\mu F)} \mathsf{fold}(y(i))\}_{i \in |A|}$ by the induction hypothesis. However, we have to construct a realiser for $\mathsf{fold}(c(y))$ itself — that is, we have to find some $b \in \mathcal{A}$, s.t. $\forall i \in |A|$ if $d \vDash_A i$, then $b \cdot d = a_i$. But there is no general construction for such a b, because there is no guarantee that the function $i \mapsto a_i$ is linear, or even computable.

Even though there might be elements without realisers in our assemblies in general, we can show by induction that elements in $|\mu F|$ have realisers, for finitary quantitative polynomial functors F, assuming of course that F is not constructed from Const_A for some A with elements without realisers. We have to restrict ourselves to finitary quantitative polynomial functors for the same reason as in the proof of Lemma 29: the induction hypothesis for $F = A \xrightarrow{1} F'$ only gives us a collection of realisers, but no computable function that selects one for every realiser of $a \in |A|$.

▶ **Proposition 30.** *Let F be a finitary quantitative polynomial functor such that if F was generated using a Const_A rule, then every $x \in |A|$ has a realiser. Then every element $x \in |\mu F|$ has a realiser.* ◀

Altogether, we have now shown how to interpret the type μF as an assembly by making use of the initial algebra of $\tilde{F}$ in $\mathcal{S}et$, and how both the algebra map $c : F(\mu F) \to \mu F$ and the unique mediating morphism $\mathsf{fold} : \mu F \to X$ are realisable, the latter if F is finitary. Furthermore, if built from components where every element has a realiser, the initial algebra will retain this property, which is often important for reasoning about terms in the model. Hence initial algebras of finitary quantitative polynomial functors are supported in the realisability model $\mathcal{M}_r$.

6 Conclusions and Future Work

We have given the first principled account of data types in quantitative type theory. This is necessary, since many established facts about ordinary data types, such as the universal applicability of containers to represent all strictly positive type formers, do not carry over to the quantitative setting. Instead we have considered quantitative polynomial functors inductively generated by a grammar. By firstly reducing elimination rules to initiality, and then concretely constructing initial algebras in the model, we have shown how finitary data types can be given semantics in a linear realisability model of QTT. This gives a precise mathematical meaning to a subset of the data types that can be defined in Idris 2, and reassurance that they are canonical, in the sense that they satisfy the universal property of initiality.

The equivalence between elimination rules and initiality works for arbitrary quantitative polynomial functors, but our proof of the existence of initial algebras in the realisability model is restricted to finitary quantitative polynomial functors. We conjecture that also infinitary data types are supported in this model, but this result is out of reach of our current proof method. Our data types are also functors on categories of *closed* types, and correspondingly we only derive induction principles in empty contexts; it would be good to relax this restriction. y Going beyond simple polynomial functors, we believe it should be possible to adapt Gambino and Hyland's reduction of the existence of initial algebras of indexed containers to the existence of initial algebras of containers [18] to the quantitative setting, at least with an extensional identity type. It seems as if the proof should extend to quantitative polynomial functors as well, with some more work. However going further, Kaposi and Kovács have had great success describing and modelling quotient and higher inductive-inductive types using a notion of signature based on internal categories with families to model the intricate dependencies between different constructors [27, 28, 26]. It would be interesting to see if a similar approach of internal type theory [14] using *quantitative* categories with families could also work to describe more expressive data types in the setting of QTT.

References

1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew D. Gordon, editor, *Foundations of Software Science and Computation Structures*, pages 23–38. Springer, 2003. doi:10.1007/3-540-36576-1_2.

2 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005. doi:10.1016/j.tcs.2005.06.002.

3 Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–28, 2020. doi:10.1145/3408972.

4 Samson Abramsky, Esfandiar Haghverdi, and Philip Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002. doi:10.1017/S0960129502003730.

5 Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 293–310. Springer, Heidelberg, 2018. doi:10.1007/978-3-319-89366-2_16.

6 Robert Atkey. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '18*, pages 56–65. ACM Press, 2018. doi:10.1145/3209108.3209189.

7 Steve Awodey, Nicola Gambino, and Kristina Sojakova. Homotopy-initial algebras in type theory. *Journal of the ACM*, 63(6), 2017. doi:10.1145/3006383.

8 Edwin Brady. Idris 2: Quantitative Type Theory in Practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*,

volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ECOOP.2021.9`.

9 Iliano Cervesato and Frank Pfenning. A Linear Logical Framework. *Information and Computation*, 179(1):19–75, 2002. `doi:10.1006/inco.2001.2951`.

10 Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. A graded dependent type system with a usage-aware semantics. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021. `doi:10.1145/3434331`.

11 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TYPES.2015.5`.

12 Thierry Coquand, Simon Huber, and Anders Mörtberg. On higher inductive types in cubical type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 255–264. Association for Computing Machinery, 2018. `doi:10.1145/3209108.3209197`.

13 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 50–66. Springer, 1990. `doi:10.1007/3-540-52335-9_47`.

14 Peter Dybjer. Internal type theory. In *Lecture Notes in Computer Science*, volume 1158 LNCS, pages 120–134. Springer, Berlin, Heidelberg, 1996. `doi:10.1007/3-540-61780-9_66`.

15 Peter Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical Computer Science*, 176(1):329–335, 1997. `doi:10.1016/S0304-3975(96)00145-4`.

16 Peter Dybjer and Anton Setzer. Induction–recursion and initial algebras. *Annals of Pure and Applied Logic*, 124(1):1–47, 2003. `doi:10.1016/S0168-0072(02)00096-9`.

17 Peng Fu, Kohei Kishida, and Peter Selinger. Linear Dependent Type Theory for Quantum Programming Languages: Extended Abstract. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2020: Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 440–453. Association for Computing Machinery, 2020. `doi:10.1145/3373718.3394765`.

18 Nicola Gambino and Martin Hyland. Wellfounded Trees and Dependent Polynomial Functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 210–225. Springer, Berlin, Heidelberg, 2003. `doi:10.1007/978-3-540-24849-1_14`.

19 Nicola Gambino and Joachim Kock. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society*, 154(1):153–192, 2013. `doi:10.1017/S0305004112000394`.

20 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987. `doi:10.1016/0304-3975(87)90045-4`.

21 Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, 137(1-3):189–239, 2006. `doi:10.1016/j.apal.2005.05.022`.

22 Peter Hancock and Anton Setzer. Interactive Programs in Dependent Type Theory. In Peter Clote and Helmut Schwichtenberg, editors, *Computer Science Logic, CSL 2000*, volume 1862 of *Lecture Notes in Computer Science*, pages 317–331. Springer, Berlin, Heidelberg, 2000. `doi:10.1007/3-540-44622-2_21`.

23 Claudio Hermida and Bart Jacobs. Structural Induction and Coinduction in a Fibrational Setting. *Information and Computation*, 145(2):107–152, 1998. `doi:10.1006/inco.1998.2725`.

24 Naohiko Hoshino. Linear Realizability. In *Computer Science Logic*, volume 4646 LNCS, pages 420–434. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 2007. `doi:10.1007/978-3-540-74915-8_32`.

25 Jasper Hugunin. Why not W? In *26th International Conference on Types for Proofs and Programs (TYPES 2020)*, 2020.

26 Ambrus Kaposi and András Kovács. Signatures and induction principles for higher inductive-inductive types. *Log. Methods Comput. Sci.*, 16(1), 2020. doi:10.23638/LMCS-16(1:10)2020.

27 Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL):2:1–2:24, 2019. doi:10.1145/3290315.

28 András Kovács and Ambrus Kaposi. Large and infinitary quotient inductive-inductive types. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 648–661. ACM, 2020. doi:10.1145/3373718.3394770.

29 Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating Linear and Dependent Types. *ACM SIGPLAN Notices*, 50(1):17–30, 2015. doi:10.1145/2775051.2676969.

30 Peter Lefanu Lumsdaine and Michael Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 169(1):159–208, 2020. doi:10.1017/S030500411900015X.

31 Conor McBride. I Got Plenty o' Nuttin'. In *Lecture Notes in Computer Science*, volume 9600, pages 207–233. Springer Verlag, 2016. doi:10.1007/978-3-319-30936-1_12.

32 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):110:1–110:30, 2019. doi:10.1145/3341714.

33 Uday S. Reddy. A linear logic model of state. Manuscript, 1993.

34 Jan M. Smith. The independence of Peano's fourth axiom from Martin-Löf's type theory without universes. *Journal of Symbolic Logic*, 53(3):840–845, 1988. doi:10.2307/2274575.

35 Tomáš Svoboda. Additive pairs in quantitative type theory. Master thesis, Charles University Prague, 2021. doi:20.500.11956/127263.

36 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

37 Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990.