Abstraction Functions as Types

Modular Verification of Algorithms and Data Structures

HARRISON GRODIN, RUNMING LI, and ROBERT HARPER, Carnegie Mellon University, USA

Modular development of programs relies on the principle that library code may be freely replaced without affecting client behavior. While an interface mediating this interaction should require a precise behavior of its implementations, allowing for downstream verification of client code, it should do so in a manner that allows private algorithmic and representation choices to vary freely. In this work we demonstrate how such modularity can be achieved in dependent type theory using a phase distinction between private algorithmic content and public client-facing behavior. We observe that modalities associated with such a phase distinction and their corresponding theorems, particularly noninterference and fracture, give rise to precise descriptions of common constructions surrounding algorithms and data structures. Using a modal construction to classify types that sufficiently restrict client-facing behavior, we use the noninterference property for the phase to state and prove a modularity property guaranteeing that implementations may be freely replaced without affecting behavior. We then cast the fracture property in the light of abstraction functions, showing internally that every type consists of a private algorithmic component, a public behavioral component representing an abstract data type, and an abstraction function between them that is uniformly activated by the behavioral phase for streamlined verification of client correctness. Finally, we use phased quotient types to ergonomically mark private data for behavioral deletion. We situate this development in a univalent adaptation of Calf, a dependent type theory for cost analysis, in order to amplify these points: beyond hiding private implementation details, we treat cost as a private matter that may be varied freely without affecting the behavior of clients.

CCS Concepts: • **Theory of computation** → **Type theory**; **Logic and verification**; *Program analysis*; *Categorical semantics*; • **Software and its engineering** → *Functional languages.*

Additional Key Words and Phrases: phase distinction, noninterference, information flow, modal type theory, abstraction, abstraction function, abstract data type, data structure, algorithm, equational reasoning, verification, mechanized proof, proof assistants

1 INTRODUCTION

Software development is fundamentally a community effort, spread over space and time, with no one person having a complete mastery of the code involved. Therefore, the single most effective tool in software development is the composition of programs from reusable, replaceable parts that are not under the control or influence of any one developer. Developers rely on abstractions—some form of *type*—to describe the assumptions that clients may rely upon when using a component, and, correspondingly, the obligations that implementors must provide. The essence of modularity is that implementations of the same abstraction may be freely swapped out without any impact on the behavior of the surrounding program.

Consider one such abstraction, the queue abstract data type $\textsc{Queue}$, and suppose some client code $f : \textsc{Queue} \to X$ implements a type X using its input queue implementation as library code. The essence of modularity is the guarantee that queue implementations can be freely replaced without affecting the functional behavior of f:

$$\text{for all } q, q' : \textsc{Queue}, \text{ we have that } f(q) = f(q').$$

Assuming that there is some basic implementation, $q_0 : \textsc{Queue}$, this description amounts to saying that the type $\textsc{Queue}$ is contractible (*i.e.*, singleton, terminal, or equivalent to the unit type). This

Authors' address: Harrison Grodin, hgrodin@cs.cmu.edu; Runming Li, runmingl@cs.cmu.edu; Robert Harper, rwh@cs.cmu.edu, Carnegie Mellon University, Computer Science Department, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA.

idea is harmonious with the intuitive understanding a client should have about queues: the behavior should be completely pinned down. We could achieve such a contractible type by defining

$$\textsc{Queue} \coloneqq \sum_{q:\textsc{PreQueue}} q = q_0,$$

where PreQueue describes a type equipped with queue operations and the constraint $q = q_0$ restricts the type and operations in q to match the behavior mandated by the specification q_0.

Say the specification q_0 has representation type $\textsc{List}\ \mathbb{N}$, a canonical form for queues. Then, other implementations would have to come equipped with a proof that their implementation type is equivalent to $\textsc{List}\ \mathbb{N}$, and the queue operations cohere according to this equivalence. The cleverness of an implementation typically serves to improve upon the efficiency of the naive code given as the behavioral specification q_0. While the type $\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}$ used to implement batched queues does not immediately satisfy this property, Angiuli et al. [4] demonstrate how to use quotient types in a univalent setting to identify states that induce equivalent client behavior. For example, we could quotient the type $\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}$ by equivalence under the function $\mathit{revAppend} : \textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N} \to \textsc{List}\ \mathbb{N}$ which converts a pair of lists representing a queue to their single-list correspondent.

Although the use of contractible types as specifications achieves modularity, it is at the expense of concerns about efficiency. The identification of data with equivalent functional behavior is pervasive: one cannot hope to uniformly extract the code that does not depend on its presence. However, the purpose of an implementation other than the specification is *efficiency*: although its behavior is fixed, the cleverness of its implementation serves to improve upon the naive code given as the behavioral specification. It is paramount, therefore, to retain *differences* between algorithms and data structures that implement a given behavior: although a queue can be thought of as a list of elements when analyzing behavior, a more clever representation (*e.g.*, a pair of lists without a quotient) may be preferable internally.

To mediate this tension between efficient code and behavioral verification, we introduce a synthetic *phase distinction* [43] between private *algorithmic* choices and public *behavior*. If we then relax the requirement on Queue, asking only for it to be *behaviorally* contractible, we allow the retention of algorithmic content that differs between queue implementations even though equivalent states must be *behaviorally* identified. We then get a weakened property that guarantees modularity of behavior:

$$\text{for all } q, q' : \textsc{Queue}, \text{ we have that } f(q) \doteq f(q'),$$

where $x \doteq x'$ denotes *behavioral* equivalence. To implement queues using a pair of lists, we may then choose to activate the quotient only in the behavioral phase.

Technically, this phase is realized as a proposition $\P_{\mathsf{beh}}$ in dependent type theory corresponding to the assumption of operating with concern only for functional behavior. The proposition gives rise to various devices in type theory, including a *behavioral modality* for entering the behavioral phase, an *algorithmic modality* for marking data as behaviorally-irrelevant [38]. The modularity property can be proved from these modalities within the type theory.

The use of a phase distinction allows us to integrate both algorithmic and behavioral concerns into a single program, justified by the *fracture theorem* [38, §3.4]. This theorem says that every type can be fractured into (or reconstructed from) three parts: a public behavioral type $X_\circ$, a private algorithmic type $X_\bullet$, and an *abstraction function* [23] $\alpha : X_\bullet \to X_\circ$ that converts the concrete representation of type $X_\bullet$ to the abstract behavior of type $X_\circ$. In other words, every representation type inherently comes equipped have a conversion to its behavioral representation type, concisely stated by the following slogan:

Types are abstraction functions.

Semantically, every type may be interpreted as a presheaf that incorporates both algorithmic and behavioral content, which can be constructed synthetically within type theory using the fracture theorem. For example, we will arrange for the presheaf

$$\left(\begin{tikzcd} \textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N} \arrow[d, "revAppend"] \\ \textsc{List}\ \mathbb{N} \end{tikzcd}\right)$$

to be the semantics of the type implementing batched queues, storing a pair of lists as the efficient algorithmic representation alongside the single-list behavioral perspective. The included *revAppend* function will fire automatically in the behavioral phase, irreversibly fusing the pair of lists into the simpler single-list representation for behavioral verification.

We situate our story in Calf, a type theory for synthetic cost analysis, which advocates for the use of such a phase under which cost annotations are erased [31]. Our development builds on this perspective, continuing to treat cost annotations as private *algorithmic* data although extending the use case of behaviorally contractible types to classify types that provide a sufficiently descriptive behavioral verification. We demonstrate the need for a more liberal use of the algorithmic modality and related erasure techniques: if implementations are to meet a unique behavior specified by an algorithmic interface, we show that the types involved must explicitly mark their own private data for redaction. We now introduce these ideas formally, recalling key elements of the phased Calf type theory and theorems about the phase modalities.

1.1 Effectful dependent type theory

To support effectful programs in dependent type theory, we operate in a dependent type theory that distinguishes between running computations and inert values, both in terms and types. We work in a version of call-by-push-value [26], inspired by Ahman et al. [3], Krishnaswami et al. [25], Pédrot and Tabareau [35], Vákár [48]. We include features of the Enriched Effect Calculus [12, 13], adapted to the dependent setting. Additionally, we use the linear universe structure of Krishnaswami et al. [25] (with levels omitted for simplicity), writing $\mathcal{V}$ for the universe of value types and C for the universe of computation types (both of which are, themselves, value types).

$$\begin{array}{rl} \text{Val.} & X, Y, Z ::= \mathbf{U}(A) \mid \mathbb{N} \mid \textsc{List}\ X \mid 1 \mid \sum_{x:X} Y(x) \mid \prod_{x:X} Y(x) \mid A \multimap B \mid \mathcal{V} \mid C \mid x =_X x' \\ \text{Comp.} & A, B, C ::= \mathbf{F}(X) \mid \prod_{x:X} A(x) \end{array}$$

We freely use standard abbreviations and notations for readability, such as $X \rightharpoonup A := \prod_{x:X} A$. We use common notations for programs involving these types, such as pattern matching on $\mathbf{ret}(x)$ (the introduction form for the $\mathbf{F}(X)$ type) as the elimination form for $\mathbf{F}(X)$. Following Calf [31], we also use the "less bureaucratic" form of adjoint type theory in which the introduction and elimination forms for the $\mathbf{U}(A)$ type are left implicit.

In Section 4 we assume commutativity of the effects present and use the monoidal structure $(\top, \otimes)$ given by linear/non-linear type theory [6, 25]:

$$A, B, C ::= \cdots \mid \top \mid A \otimes B$$

1.2 Univalent equality

This work takes place in a univalent type theory [37, 47] equipped with proof-relevant extensional equality $x =_X x'$ (*i.e.*, path types). We assume the following extensionality principles for equality:

$$f =_{\prod_{x:X} Y(x)} f' :\equiv \prod_{x:X} f(x) =_{Y(x)} f'(x) \tag{1a}$$

$$(x, y) =_{\sum_{x:X} Y(x)} (x', y') :\equiv \sum_{p:x=_X x'} p_*(y) =_{Y(x')} y' \tag{1b}$$

$$X =_{\mathcal{V}} X' :\equiv \sum_{f:X\to X'} \mathsf{isEquiv}_{\mathcal{V}}(f) \tag{1c}$$

$$A =_{\mathcal{C}} A' :\equiv \sum_{f:A\multimap A'} \mathsf{isEquiv}_{\mathcal{C}}(f) \tag{1d}$$

In a univalent setting equality of dependent products and sums are given extensionally. Moreover, the principle of univalence says that equality at the universes of value types $\mathcal{V}$ and computation types $\mathcal{C}$ is equivalence [37, Chapter 9], using linear functions as maps between computation types [12, 13], where

$$\mathsf{isEquiv}_{\mathcal{V}}(f : X \to X') :\equiv \sum_{s,r:X'\to X} (s \mathbin{;} f = \mathrm{id}_{X'}) \times (f \mathbin{;} r = \mathrm{id}_X)$$

$$\mathsf{isEquiv}_{\mathcal{C}}(f : A \multimap A') :\equiv \sum_{s,r:A'\multimap A} (s \mathbin{;} f = \mathrm{id}_{A'}) \times (f \mathbin{;} r = \mathrm{id}_A)$$

say that f is an equivalence when it admits both a section and a retraction.

Although we are working in a univalent setting, the types we consider will all be set-truncated (aside from the universes $\mathcal{V}$ and $\mathcal{C}$). Thus, when we define a higher inductive type (in this work, only quotient types), we omit explicit set-truncation for brevity.

1.3 Behavioral phase distinction

To facilitate the development proposed above, we postulate a synthetic *phase distinction* [41, 43] that allows the uniform isolation of the behavioral semantics of programs. Specifically, the *behavioral phase*[1] is a postulated proposition $\P_{\mathsf{beh}} : \mathcal{V}$ that, when inhabited, erases details relevant only for efficiency, leaving behind only the behavior for analysis of correctness. Because $\P_{\mathsf{beh}}$ is a proposition, we always use variable name $\_ : \P_{\mathsf{beh}}$ for convenience.

1.3.1 Behavioral and algorithmic modalities. Associated with the proposition $\P_{\mathsf{beh}}$ are a pair of idempotent monadic modalities, the behavioral modality and the algorithmic modality [38].

Definition 1.1. The *behavioral modality* $\bigcirc X :\equiv \P_{\mathsf{beh}} \to X$ is the reader monad for the type $\P_{\mathsf{beh}}$, imposing the behavioral phase to suppress cost information and isolate the behavioral aspect of a type X. We say a type X is *behavioral* when $\bigcirc X = X$,[2] and we call an inhabitant of a behavioral type a *behavior*. We say a type family $Y : X \to \mathcal{V}$ is behavioral when $Y(x)$ is behavioral for all $x : X$, and we say a map $f : Y \to X$ is behavioral when $\mathsf{fib}_f(x) :\equiv \sum_{y:Y} f(x) = y$ is behavioral.

[1]We use the "behavioral/algorithmic" terminology in place of the "extensional/intensional" terminology from previous developments of Calf [20, 31], avoiding confusion with the unrelated ideas of extensional/intensional type theory and mathematical extensionality principles while emphasizing the abstraction viewpoint of this work.

[2]In other work this notion has been referred to as open-modal/$\bigcirc$-modal [38], $\P_{\mathsf{beh}}$-transparent [42, 45], and (purely) extensional [20, 31] types.

We may unambiguously use the same notation for a dependent variant of the behavioral modality, as well: given a type family $X : ¶_{\mathsf{beh}} \to \mathcal{V}$, we write $\bigcirc X$ for $\prod_{\_:¶_{\mathsf{beh}}} X(\_)$. Additionally, we write $x \mathrel{\mathring{=}} x'$ as a shorthand for $\bigcirc(x = x')$.

Definition 1.2. The *algorithmic modality* $\bullet X$ marks a type as behaviorally irrelevant: crucially, we have that $\bigcirc\bullet X = 1$. It is defined as the pushout of the behavioral equivalence $\pi_1 : X \times ¶_{\mathsf{beh}} \to X$ along the map $\pi_2 : X \times ¶_{\mathsf{beh}} \to ¶_{\mathsf{beh}}$, sometimes written $\bullet X := X \vee ¶_{\mathsf{beh}}$:

$$\begin{tikzcd} X \times ¶_{\mathsf{beh}} \arrow[r, "\pi_2"] \arrow[d, "\pi_1"] & ¶_{\mathsf{beh}} \arrow[d, "*"] \\ X \arrow[r, "\eta^\bullet"] \arrow[ur, phantom, "\ulcorner", very near end] & \bullet X \end{tikzcd}$$

$$\begin{aligned} &\mathbf{data}\ \bullet\ (X : \mathcal{V}) : \mathcal{V}\ \mathbf{where} \\ &\quad \eta^\bullet : X \to \bullet X \\ &\quad * : \{\_ : ¶_{\mathsf{beh}}\} \to \bullet X \\ &\quad \_ : (x : X)\ \{\_ : ¶_{\mathsf{beh}}\} \to \eta^\bullet x = * \end{aligned}$$

For convenience, we make the argument to the constructor $*$ of type $¶_{\mathsf{beh}}$ implicit, indicated with braces. The quotient case induced by the pushout must be respected by users of this modality: when casing on data of type $\bullet X$, both the $\eta^\bullet$ and $*$ cases must agree (behaviorally, because $*$ may only be constructed assuming $¶_{\mathsf{beh}}$ holds). We say a type X is *algorithmic* when $\bullet X = X$,[3] and we call an inhabitant of an algorithmic type an *algorithm*. We say a type family $Y : X \to \mathcal{V}$ is algorithmic when $Y(x)$ is algorithmic for all $x : X$, and we say a map $f : Y \to X$ is algorithmic when $\mathsf{fib}_f(-)$ is algorithmic.

Lemma 1.3. *Algorithmic data can be characterized in terms of their behavior:*

(1) *a type X is algorithmic exactly when $\bigcirc X$ is contractible [38, Example 1.31], and*
(2) *a map $f : Y \to X$ is algorithmic exactly when $\bigcirc f : \bigcirc Y \to \bigcirc X$ is an equivalence [38, Lemma 1.35 and Theorem 3.1].*

Algorithmic types will be a central notion in this work, describing classes of algorithms that all share a single behavior. Although an algorithmic type may have many inhabitants, they must all implement the same behavior, rendering the type behaviorally trivial. For an algorithmic type X, we have that $\bigcirc X$ is contractible; we refer to center of contraction $x_0 : \bigcirc X$ as the *behavior* of X, because all algorithms of type X collapse to x_0 under the aegis of the behavioral phase.

1.3.2 Noninterference and modularity. These modalities admit the statement and proof of a *noninterference theorem*, which states that the algorithmic content of an implementation does not impact the behavior of its clients. This ensures that implementations may be freely substituted for one another without changing client behavior, even though algorithmic-level details about the client (such as cost) may be impacted.

Theorem 1.4 (Noninterference [31]). *Let X be an algorithmic type, and let Y be an arbitrary type. Then, the function space $X \to Y$ is behaviorally equivalent to Y.*

The essence of this theorem is that behaviorally, x_0 is the unique input of type X (up to equivalence). This means that given a program of type Y using an algorithm of type X, we may choose *any* convenient implementation when verifying behavioral correctness.

Corollary 1.5 (Modularity). *Let X be an algorithmic type, and let Y be an arbitrary type. Then, for all $f : X \to Y$ and $x, x' : X$, we have that behaviorally, $f(x) = f(x')$.*

In other words: for an implementation f of Y depending on an algorithmic type X, we may freely swap any $x : X$ for any $x' : X$ and guarantee identical behavior.

[3] In other work this notion has been referred to as closed-modal/$\bullet$-modal [38], $\bigcirc$-connected [38], $¶_{\mathsf{beh}}$-sealed [42, 44, 45], and (purely) intensional [20, 31] types.

Remark 1.6 (Security). Modularity and the behavioral phase may be viewed through the lens of security, where the behavioral phase corresponds to a public, low-security environment and the default algorithmic phase corresponds to a private, high-security environment [45]. By default, we operate in the private environment, capable of writing secret implementation details pertaining to cost and clever implementation. When we switch to the public environment, though, implementation details (including cost) are redacted. It is in this sense that the behavioral phase guarantees a notion of abstraction and modularity: private data is guaranteed to be redacted in the public phase and may therefore be swapped with any other private data at will with no impact. ⌟

1.3.3 Fracture and gluing. Importantly, every type consists of a behavioral component, an algorithmic component, and a function mapping the latter to the former, understood in this work as an abstraction function.

Theorem 1.7 (Fracture and Gluing [38]). *Let $\mathcal{V}$ be the universe of (value) types, and let $\mathcal{V}_\circ$ and $\mathcal{V}_\bullet$ be the universes of behavioral and algorithmic types, respectively. There is an equivalence*

$$\mathcal{V} = \sum_{X_\circ : \mathcal{V}_\circ} \sum_{X_\bullet : \mathcal{V}_\bullet} X_\bullet \to \bullet X_\circ$$

between types $X : \mathcal{V}$ and their fracturing into a behavioral type $X_\circ$, an algorithmic type $X_\bullet$, and an abstraction function $\alpha : X_\bullet \to \bullet X_\circ$.

Proof Sketch. We give an isomorphism explicitly. In the forward direction, *fracture* the type X by sending it to the triplet $(\bigcirc X, \bullet X, \bullet \eta^\circ_X)$. In the reverse direction, *glue* the parts $(X_\circ, X_\bullet, \alpha)$ by sending them to the following pullback, which we denote $\mathsf{Glue}(X_\circ, X_\bullet, \alpha)$:

$$\begin{tikzcd}
\mathsf{Glue}(X_\circ, X_\bullet, \alpha) \arrow[r] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & X_\bullet \arrow[d, "\alpha"] \\
X_\circ \arrow[r, "\eta^\bullet_{X_\circ}"] & \bullet X_\circ
\end{tikzcd}$$

$$\mathsf{Glue}(X_\circ, X_\bullet, \alpha) :\equiv \sum_{x_\circ : X_\circ} \sum_{x_\bullet : X_\bullet} \eta^\bullet_{X_\circ} x_\circ = \alpha x_\bullet$$

The round-trip condition on $\mathcal{V}$ says that every type $X : \mathcal{V}$ can be recovered by the following pullback of $\bigcirc X$ and $\bullet X$:

$$\begin{tikzcd}
X \arrow[r, "\eta^\bullet_X"] \arrow[d, "\eta^\circ_X"] \arrow[dr, phantom, very near start, "\lrcorner"] & \bullet X \arrow[d, "\bullet \eta^\circ_X"] \\
\bigcirc X \arrow[r, "\eta^\bullet_{\bigcirc X}"] & \bullet \bigcirc X
\end{tikzcd}$$

$$X = \sum_{x_\circ : \bigcirc X} \sum_{x_\bullet : \bullet X} \eta^\bullet_{\bigcirc X} x_\circ = \bullet \eta^\circ_X x_\bullet$$

The other round-trip condition ensures that $\bigcirc \mathsf{Glue}(X_\circ, X_\bullet, \alpha) = X_\circ$ and $\bullet \mathsf{Glue}(X_\circ, X_\bullet, \alpha) = X_\bullet$. □

1.3.4 Semantics. We recall three important semantic models of Calf, differing centrally in their interpretation of the behavioral phase proposition, $\P_{\mathsf{beh}}$.

Semantics 1 (Behavioral). We may interpret $\P_{\mathsf{beh}}$ as the true proposition. Because the assumption that programs exist only for their behavior is true, programs collapse to their behavioral or mathematical interpretations. In this setting $\bigcirc X = X$, $\bullet X = 1$, and $\mathsf{Glue}(X_\circ, X_\bullet, \alpha) = X_\circ$.

Semantics 2 (Algorithmic). We may interpret $\P_{\mathsf{beh}}$ as the false proposition. Because the assumption that programs exist only for their behavior is false, all behavioral assumptions are erased, leaving behind the standard algorithmic implementation. In this setting $\bigcirc X = 1$, $\bullet X = X$, and $\mathsf{Glue}(X_\circ, X_\bullet, \alpha) = X_\bullet$.

Semantics 3 (Presheaf). Calf can be given a Kripke semantics in the Sierpinski topos, *i.e.* presheaves on the "walking arrow" category $\mathbb{2} := \{\circ \to \bullet\}$ [31, §5]. Each type X is interpreted as a presheaf $[\![X]\!]$, where $[\![X]\!]_\bullet$ is the algorithmic component and $[\![X]\!]_\circ$ is the behavioral component. The induced map $[\![X]\!]_\bullet \to [\![X]\!]_\circ$ can be thought of as an abstraction function, converting the private, algorithmic representation $[\![X]\!]_\bullet$ to the public, behavioral representation $[\![X]\!]_\circ$. For example:

$$[\![X]\!] := \left(\begin{tikzcd}[\![X]\!]_\bullet \arrow[d] \\ [\![X]\!]_\circ\end{tikzcd}\right) \qquad [\![\mathbb{N}]\!] := \left(\begin{tikzcd}\mathbb{N} \arrow[d, "\mathrm{id}"] \\ \mathbb{N}\end{tikzcd}\right) \qquad [\![\P_{\mathsf{beh}}]\!] := \text{よ}(\circ) = \left(\begin{tikzcd}0 \arrow[d] \\ 1\end{tikzcd}\right)$$

Standard base types, such as $\mathbb{N}$, are interpreted as constant presheaves, with identical algorithmic and behavioral components and the identity function between them. The behavioral phase $\P_{\mathsf{beh}}$ is defined as the Yoneda embedding of $\circ : \mathbb{2}$, with an empty algorithmic component and a trivial behavioral component, combining Semantics 1 and 2. The modalities are interpreted as follows:

$$[\![\bigcirc X]\!] = [\![\P_{\mathsf{beh}} \to X]\!] = \left(\begin{tikzcd}[\![X]\!]_\circ \arrow[d, "\mathrm{id}"] \\ [\![X]\!]_\circ\end{tikzcd}\right) \qquad [\![\bullet X]\!] = [\![X \vee \P_{\mathsf{beh}}]\!] = \left(\begin{tikzcd}[\![X]\!]_\bullet \arrow[d] \\ 1\end{tikzcd}\right)$$

Notice that a type X is behavioral when it is interpreted as a constant presheaf with $[\![X]\!]_\bullet = [\![X]\!]_\circ$ and algorithmic when it has $[\![X]\!]_\circ = 1$. In other words a type is behavioral when its intrinsic abstraction function is an equivalence, and a type is algorithmic when its intrinsic abstraction function is a unique map into 1 that fully erases information. Gluing can be viewed as synthetically constructing a presheaf,

$$[\![\mathsf{Glue}(X_\circ, X_\bullet, \alpha)]\!] = \left(\begin{tikzcd}[\![X_\bullet]\!]_\bullet \arrow[d, "{[\![\alpha]\!]_\bullet}"] \\ [\![X_\circ]\!]_\bullet\end{tikzcd}\right),$$

where $[\![\alpha]\!] : [\![X_\bullet]\!] \to [\![\bullet X_\circ]\!]$ is a map of presheaves. In this sense every type is interpreted as an abstraction function, describing (the algorithmic parts of) types $X_\bullet$ and $X_\circ$ with an associated abstraction function to erase private details.

1.4 Cost as an effect

In Calf cost is treated abstractly as an effect: we write $\mathbf{charge}\langle c\rangle(a)$ for a print-like effect that records $c : \mathbb{C}$ units of abstract cost before running the computation $a : A$, where $\mathbb{C}$ is a value type representing cost equipped with a monoid structure $(0, +)$.[4] The cost effect is governed by the following laws, using the monoid operations associated with cost algebra $\mathbb{C}$.

Axiom 1.8. *The cost effect respects the monoid* $(\mathbb{C}, 0, +)$*:*

$$\mathbf{charge}\langle 0\rangle(a) = a \tag{2a}$$

$$\mathbf{charge}\langle c_1\rangle(\mathbf{charge}\langle c_2\rangle(a)) = \mathbf{charge}\langle c_1 + c_2\rangle(a) \tag{2b}$$

In Calf it is crucial that the cost algebra $\mathbb{C}$ be algorithmic: this allows the cost effect to be omitted in the behavioral phase, supporting reasoning about correctness without involving their cost.

[4]We alter the notation $\mathsf{step}^c(-)$ from previous developments in Calf, emphasizing that c is counting an abstract notion of cost, not the number of evaluation steps.

Axiom 1.9. *The cost algebra $\mathbb{C}$ is algorithmic.*

Theorem 1.10. *If $¶_{\mathsf{beh}}$ holds, then for all $c : \mathbb{C}$, we have* $\mathbf{charge}\langle c \rangle(e) = e$.

Proof. Suppose $¶_{\mathsf{beh}}$ holds, and let $c : \mathbb{C}$ be arbitrary. Then, because $\mathbb{C}$ is algorithmic, it is contractible. Therefore, all elements of $\mathbb{C}$ are equal, so $c =_{\mathbb{C}} 0$. The result follows by Eq. (2a). □

Remark 1.11. Although the presence of cost amplifies the importance of certain issues in this paper, the techniques developed here for modular verification of algorithms and data structures stand freely without cost annotations, as well. In particular note that the trivial monoid is a valid cost model, as $\mathbb{C} :\equiv 1$ is algorithmic. ⌟

In this paper, as in Calf [31], we choose to define $\mathbb{C} :\equiv \bullet\mathbb{N}$ for the purpose of our examples. For convenience, we avoid $\eta^{\bullet}$ when constructing costs, and we avoid the use of $*$ in favor of 0.

1.5 Contributions

In this paper we identify algorithmic types as a novel source of semantic modularity suitable for the compositional verification of behavior, elevating the concept of algorithmicity from merely a tool for cost erasure to a more general programmer-facing construct for guaranteeing the client-facing redaction of choices made for the purpose of efficiency. We demonstrate sources of algorithmic types that specify common algorithms and data structures, using the fracture/gluing property and behavioral quotients to mark algorithmic details for erasure in the behavioral phase, guaranteeing that algorithms and data structures may be swapped out interchangeably without affecting behavior. This perspective lends itself to a simple and precise consolidation of many concepts about algorithms and data structures, including benign effects, abstraction functions, relational parametricity, views, and smart constructors.

Synopsis. This paper is organized as follows. In Section 2 we consider the specification of sorting algorithms to justify the claim that algorithmic types are a sensible notion for the specification of algorithmic problems, and we consider subtleties that arise with uniqueness of outputs and modularity. In Section 3 we expand the story to data structures and abstract data types, considering an algorithmic type specifying persistent queues and implementing the representation type as a synthetic abstraction function via gluing. In Section 4 we present an ergonomic approach to specifying synthetic abstraction functions implicitly using behavioral quotients to redact efficiency-only aspects of representation types when verifying behavior of client code. Finally, in Section 5 we summarize results, connect to related work, and suggest directions for future development.

2 ALGORITHMIC TYPES AS BEHAVIORAL SPECIFICATIONS

To specify an algorithmic problem, it is typical to make precise the intended behavior that the corresponding algorithms must implement. Algorithms may then implement the given behavior with various cost characteristics. This notion is made precise via algorithmic types: although an algorithmic type may have many inhabitants, it must behaviorally be a singleton, contracting down to the single guaranteed behavior required by the specification.

2.1 Constructing algorithmic types

The canonical strategy for constructing an algorithmic type out of an arbitrary type X consists of providing a behavior $x_0 : \bigcirc X$ and then considering the subtype of X consisting of all $x : X$ that behaviorally match x_0, which we write as $\{X \mid \P_{\mathsf{beh}} \hookrightarrow x_0(\_)\}$[5]:

$$\{X \mid \P_{\mathsf{beh}} \hookrightarrow x_0(\_)\} :\equiv \sum_{x:X} x \doteq x_0(\_)$$

This type is always algorithmic, with behavior

$$\lambda(\_ : \P_{\mathsf{beh}}).\ (x_0(\_), \lambda(\_ : \P_{\mathsf{beh}}).\ \mathsf{refl}) : \bigcirc\{X \mid \P_{\mathsf{beh}} \hookrightarrow x_0(\_)\};$$

the proof is a minor phase-sensitive adaptation of a standard argument from homotopy type theory [37, Theorem 10.1.14]. In fact any algorithmic type can be put in this form: if Y is algorithmic with behavior y_0, then $Y = \{Y \mid \P_{\mathsf{beh}} \hookrightarrow y_0\}$.

Example 2.1. To classify all sorting algorithms, we may use the algorithmic type

$$\{\mathbf{U}(\textsc{List}\ \mathbb{N} \rightharpoonup \mathbf{F}(\textsc{List}\ \mathbb{N})) \mid \P_{\mathsf{beh}} \hookrightarrow \mathit{isort}\},$$

where an element is a function $f : \mathbf{U}(\textsc{List}\ \mathbb{N} \rightharpoonup \mathbf{F}(\textsc{List}\ \mathbb{N}))$ equipped with a proof that its behavior matches insertion sort, $f \doteq \mathit{isort}$. ⌟

Although this technique does accurately describe all algorithmic types, it can obfuscate the intended verification strategy. For example, to show that merge sort or randomized quick sort is a valid sorting algorithm (*i.e.*, inhabits the type given in Example 2.1), one has to prove that $\mathit{msort} \doteq \mathit{isort}$ or $\mathit{qsort} \doteq \mathit{isort}$. Although this is true, the proof itself is structured around the idea that *isort*, *msort*, and *qsort* are all valid sorting algorithms on $\textsc{List}\ \mathbb{N}$, and all such sorting algorithms are equivalent, as verified by Niu et al. [31]. Taking this perspective, we can give a more ergonomic version of the algorithmic type of Example 2.1.

Example 2.2. Let $\textsc{IsSorted}(l)$ be a propositional type family over $\textsc{List}\ \mathbb{N}$ that is inhabited exactly when list l is sorted, and let $\textsc{IsPerm}(l, l')$ be a propositional type family over $\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}$ that is inhabited exactly when list l is a permutation of list l'.[6] For every list l, there exists a unique list l' such that the types $\textsc{IsSorted}(l')$ and $\textsc{IsPerm}(l, l')$ are inhabited[7]; therefore, the type

$$\textsc{SortedPerm}(l) :\equiv \textstyle\sum_{l':\textsc{List}\ \mathbb{N}} \textsc{IsSorted}(l') \times \textsc{IsPerm}(l, l')$$

is contractible (and therefore algorithmic) for all $l : \textsc{List}\ \mathbb{N}$.

Absent of any restrictions on the effects available, the type $\mathbf{U}(\mathbf{F}(\textsc{SortedPerm}(l)))$ need *not* be algorithmic; for example, if errors are available from the monad $\mathbf{U}(\mathbf{F}(-))$, an inhabitant of this type could either return or error. However, the type

$$\star\textsc{SortedPerm}(l) :\equiv \{\mathbf{U}(\mathbf{F}(\textsc{SortedPerm}(l))) \mid \P_{\mathsf{beh}} \hookrightarrow \mathbf{ret}(x_0(\_))\}$$

is algorithmic, where $x_0 : \bigcirc(\textsc{SortedPerm}(l))$ is the behavior of $\textsc{SortedPerm}(l)$. Behaviorally, an inhabitant computation $e : \mathbf{U}(\mathbf{F}(\textsc{SortedPerm}(l)))$ is restricted to be $\mathbf{ret}(x_0(\_))$, returning a value; even though e may use monadic effects, such as cost (in insertion sort and merge sort) and nondeterminism/randomization (in quick sort [20]), the effects must be *benign*, trivializing in the behavioral phase.

Abstracting over the list $l : \textsc{List}\ \mathbb{N}$ to be sorted, we get that the type

$$\textsc{Sort} :\equiv \textstyle\prod_{l:\textsc{List}\ \mathbb{N}} \star\textsc{SortedPerm}(l)$$

is algorithmic, describing all sorting algorithms. This type $\textsc{Sort}$ is equivalent to the algorithmic type of Example 2.1; however, although $\textsc{Sort}$ is also equipped with a behavior (of type $\bigcirc(\textsc{Sort})$)

[5]We use a notation inspired by extension types [36, 43]. However, we use typal equality, because the proof that an algorithm matches its behavioral reference need not be definitional.

[6]In some definitions, such as the Agda standard library [46], these type families may not be immediately propositional. In that case, one can propositionally truncate [37, §14] to obtain propositionality.

[7]The proof of this fact is a sorting *function*, without cost annotations, serving as the center of contraction.

based on the proof of algorithmicity, this fact is not explicit in the type itself, which streamlines the implementation of sorting algorithms of this type Sort. ⌟

Remark 2.3. When verifying programs in a dependent type theory, one typically faces the issue of not knowing how much needs to be proved. For example, in the case of sorting algorithms, domain-specific knowledge suggests that one should prove that the output list is sorted and a permutation of the input list. However, how can one justify that both properties are required? Because if either requirement is omitted, the resulting type fails to be algorithmic! The proposal of algorithmic types in this work provides a general theoretical foundation for making such guarantees: *one knows that an algorithm specification is precise enough when its type is algorithmic.* ⌟

The principles used in Example 2.2 are instances of more general techniques that allow us to specify an algorithmic problem via a property that its results should satisfy uniquely. First, we elaborate on the $\star$ construction.

Definition 2.4. Let X be an algorithmic type with behavior $x_0 : \bigcirc X$. We write

$$\star X := \{\mathbf{U}(\mathbf{F}(X)) \mid \P_{\mathsf{beh}} \hookrightarrow \mathbf{ret}(x_0(\_))\}$$

for the type of computations of type $\mathbf{U}(\mathbf{F}(X))$ that behaviorally return. This operator preserves algorithmicity, sending an algorithmic type X to an algorithmic type $\star X$. Moreover, $\star$ forms a monad on the universe of algorithmic types, which we refer to as the *benign effect monad* (relative to the monad $\mathbf{U}(\mathbf{F}(-))$).

The concept of benign effects slots in smoothly with the perspective that the behavioral phase erases private details: in the benign effect monad, while effects may be used for implementation purposes, a client must not observe the effects behaviorally.

For an algorithmic problem that takes inputs of type X, the valid output can be described as an algorithmic type family $Y(x)$, for each $x : X$. Then, we may use dependent products and the benign effect monad to encode an algorithmic problem as an algorithmic type, abstracting Example 2.2.

Theorem 2.5. *Let X be an arbitrary type, and let $Y : X \to \mathcal{V}$ be a family of types indexed by X such that each $Y(x)$ is algorithmic*[8]*. Then, the type $\prod_{x:X} Y(x)$ is algorithmic.*

Proof. Follows from dependent products preserving contractability [47, Lemma 3.11.6]. □

Combining Theorem 2.5 with the benign effect monad preserving algorithmicity, we recover Example 2.2, where $X := \textsc{List}\ \mathbb{N}$ and $Y(x) := \star\textsc{SortedPerm}(x)$. Variations of X and Y give other common algorithmic problem shapes.

Example 2.6. Let $P : X \to \mathcal{V}$ be a family of decidable propositions: every $P(x)$ is a proposition, and for every $x : X$, either $P(x)$ or $\neg P(x)$. Then, letting

$$Y(x) := \star(P(x) + \neg P(x)),$$

we have that $\prod_{x:X} Y(x)$ is the type of decision procedures for P. Behaviorally, there is a single inhabitant of this type that states whether $P(x)$ or $\neg P(x)$ holds. In general, though, algorithms of type $\prod_{x:X} Y(x)$ can have various cost characteristics. ⌟

Example 2.7. Let $X := \textsc{NonemptyList}\ \mathbb{N}$ be the type of nonempty lists of natural numbers, and let $\textsc{Element}\ x$ be the type of inhabitants of x. Defining

$$Y(x) := \star \textstyle\sum_{n:\textsc{Element}\ x} \prod_{n':\textsc{Element}\ x} n \leq_{\mathbb{N}} n',$$

[8]In practice $Y(x)$ will often be contractible, not merely algorithmic.

$$\begin{array}{l}\textbf{data } X/{\cong} : \mathcal{V} \textbf{ where}\\ \quad \eta : X \to X/{\cong}\\ \quad \_ : \prod_{x,x':X} x \cong_X x' \to \eta x \mathrel{\dot{=}} \eta x'\end{array}$$

(a) Behavioral quotient by preorder isomorphism.

$$\begin{array}{l}\textbf{data } \|X\|_{¶_{\mathsf{beh}}} : \mathcal{V} \textbf{ where}\\ \quad \eta : X \to \|X\|_{¶_{\mathsf{beh}}}\\ \quad \_ : \prod_{x,x':X} \eta x \mathrel{\dot{=}} \eta x'\end{array}$$

(b) Behavioral truncation.

Fig. 1. Behavioral quotients used to behaviorally ignore stability of sorting.

we have that $\prod_{x:X} Y(x)$ is the type of algorithms that find the minimum number contained in a nonempty list. Behaviorally, there is a single inhabitant of this type, although multiple algorithms may implement this type; for example, an optimized algorithm could exit early returning 0 if it is ever found in the list. ⌟

Overall, this development hinges on the idea that $Y(x)$ itself is algorithmic; in other words, there is (behaviorally) a unique answer that can be expected from every algorithm implementation. Sometimes, this is not the case, though: multiple solutions to a problem could exist. Just as we use the algorithmic modality to quotient away information in the behavioral phase, we can use other behavioral quotients to ensure algorithmicity in the case that multiple solutions are possible.

2.2 Disambiguating specifications using behavioral quotients

Sometimes, an algorithm may produce an output that is not, a priori, uniquely determined by its input. An algorithm may be allowed to produce one of many valid solutions; for example, in an arbitrary comparison-based sort, stable and unstable sorting algorithms differ on how they treat comparison-isomorphic elements. At first glance, this seems to trivialize the entire subject of algorithmic types: is it not desirable to have many possible answers? However, this fundamentally breaks modularity (Corollary 1.5): if multiple behaviors are allowed, one cannot freely swap out algorithms. For example, if client code may view the results of stable and unstable sorting algorithms differently, it could have different behavior depending on the stability. To avoid this issue, we may use behavioral quotients to explicitly identify distinct behaviors.

Example 2.8. In Example 2.2 we showed that the type of algorithms that sort a list of natural numbers is algorithmic. How can we generalize this result to sorting algorithms for an arbitrary element type X? In the implementation of sorting algorithms it is essential that the preorder on elements of type X be total. However, under only these conditions, it is not true that a sorted permutation of a list exists uniquely. For example, if $X := \mathbb{N} \times \mathtt{string}$ where comparison is performed at the first component only, then both $[(3, \texttt{"a"}), (3, \texttt{"b"})]$ and $[(3, \texttt{"b"}), (3, \texttt{"a"})]$ are sorted permutations of $[(3, \texttt{"a"}), (3, \texttt{"b"})]$. To recover algorithmicity, we must alter the types involved such that any comparison-isomorphic permutations are considered equal. Two approaches to accomplish this are given as follows:

(1) We may ask that the ordering relation on element type X is *behaviorally* antisymmetric. Say that a comparison relation $\leq_X$ is antisymmetric when the type

$$\textsc{Antisymmetric}(\leq_X) := \prod_{x,x':X} x \cong_X x' \to x = x'$$

is inhabited, where $x \cong_X x' := (x \leq_X x') \times (x' \leq_X x)$. If $\leq_X$ is behaviorally antisymmetric, then the type $\prod_{l:\textsc{List}\ X} \star\textsc{SortedPerm}(l)$ is algorithmic. Because the comparison relation $\leq_{\mathbb{N}}$ is antisymmetric (and thus behaviorally antisymmetric), this scenario directly generalizes that of Example 2.2. If the ordering relation on X is not behaviorally antisymmetric, though—such as for $X := \mathbb{N} \times \mathtt{string}$ with comparison of numbers only—we may behaviorally quotient X to identify comparison-isomorphic elements in the behavioral phase.

This new type $X/{\cong}$, shown in Fig. 1a, consists of a quotient of X by the equivalence relation $\P_{\mathsf{beh}} \times \cong$. In Semantics 1 this is simply a quotient as usual, and in Semantics 2, $X/{\cong}$ is equivalent to X because the quotient is vacuous. Now, although $(3, \texttt{"a"})$ and $(3, \texttt{"b"})$ are distinct, the injections $\eta(3, \texttt{"a"})$ and $\eta(3, \texttt{"b"})$ are equal at type $X/{\cong}$.

(2) Alternatively, we may leave the element type unmodified and instead behaviorally propositionally truncate the type of sorted permutations, using the behavioral truncation given in Fig. 1b. In Semantics 1 this is simply propositional truncation $\|X\|$, and in Semantics 2, $\|X\|_{\P_{\mathsf{beh}}}$ is equivalent to X because the quotient is vacuous. Using this truncation, the type

$$\prod_{l:\textsc{List}\ X} \star\|\textsc{SortedPerm}(l)\|_{\P_{\mathsf{beh}}}$$

is algorithmic, because differing sorted permutations are behaviorally identified.

Notice that in either case, programs using the result of a sorting algorithm must have identical behavior on identified sorted permutations. Although operating on one permutation may have differing cost compared to another, the quotients ensure that the same behavior is always encountered given comparison-equivalent permutations. ⌟

Sometimes, it is discussed whether a sorting algorithm is *stable*, preserving the original relative ordering of comparison-isomorphic elements. The perspective of Example 2.8 suggests that stability of sorting algorithms is an algorithmic property, possibly affecting efficiency but never affecting behavioral correctness. Saying that a sorting algorithm is (un)stable is akin to proving a cost bound: algorithmically, this may matter, but behaviorally, there is still only one sorting function.

Remark 2.9. If stability is desired for correctness, one could strengthen the requirements imposed by the type in exchange for avoiding the behavioral quotients. For example, one could use a stable sorting algorithm as a specification implementation, such as

$$\{\mathbf{U}(\textsc{List}\ X \rightharpoonup \mathbf{F}(\textsc{List}\ X)) \mid \P_{\mathsf{beh}} \hookrightarrow \mathit{isort}\}$$

adapted from Example 2.1. This algorithmic type describes a different class of algorithms, the “stable sorting algorithms”, as opposed to the “sorting algorithms” described in Example 2.8. When the comparison ordering $\leq_X$ is behaviorally antisymmetric, these notions coincide. ⌟

Behavioral quotients may be used in the specification of many other algorithms. For example, algorithms that find extremal solutions commonly refer to the extreme only up to some heuristic (such as the shortest path using path length, or a maximal spanning tree using tree weight); then, a behavioral quotient can ensure that the behavior of downstream code does not depend on the representative solution.

2.3 Compositional verification via modularity

By noninterference (Theorem 1.4) and modularity (Corollary 1.5), the behavioral verification of code downstream of an algorithm only depends on the unique behavior mandated by the algorithm; therefore, any algorithm implementation may be selected for convenience.

Example 2.10. Consider the following piece of downstream code, where *minimum* finds the least element of its input list:

$\mathit{downstream} : \mathbf{U}(\textsc{Sort} \rightharpoonup \textsc{List}\ \mathbb{N} \rightharpoonup \textsc{List}\ \mathbb{N})$
$\mathit{downstream}\ \mathit{sort}\ l := \mathit{sort}\ (\mathit{minimum}\ l :: l)$

We may wish to prove some facts about the behavior of *downstream* on various inputs. For example, we may wish to show that for all *sort* and l,

$$\mathit{downstream}\ \mathit{sort}\ l \doteq \mathit{minimum}\ l :: \mathit{sort}\ l.$$

One proof technique is to argue in terms of the definition of being a sorted permutation of $\mathit{minimum}\ l :: l$, which all elements of type Sort are obliged to produce. More cleverly, though, we may simply pick a sorting algorithm that makes this theorem easy to prove, such as (in this case) insertion sort. To prove the general claim, it suffices to show that

$$\mathit{downstream}\ \mathit{isort}\ l \doteq \mathit{minimum}\ l :: \mathit{isort}\ l,$$

for all l, which follows straightforwardly from the definition of insertion sort. ⌟

3 ABSTRACT DATA TYPES, DATA STRUCTURES, AND GLUING

Using the behavioral phase, we may classify a type equipped with operations that restricts to a known behavior from within the phase, expressing the notion of an *abstract data type*. The informal definition of an abstract data type is well-known:

> In computer science an *abstract data type (ADT)* is a mathematical model for data types, defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. [1]

To describe an abstract data type, it is essential to describe the behavior a client can expect from its data structure implementations, which can be reified as a type-level requirement using the behavioral phase. For example, consider the following "signature" PreQueue:

$$\textsc{PreQueue} \coloneqq \sum_{X:\mathcal{V}} (\mathsf{empty} : X) \times (\mathsf{enqueue} : \mathbf{U}(X \rightharpoonup \mathbb{N} \rightharpoonup \mathbf{F}(X))) \times (\mathsf{dequeue} : \mathbf{U}(X \rightharpoonup \mathbf{F}(\mathbb{N} \times X)))$$

A reader might intuit from the included labels that this interface describes an abstract data type that contains natural numbers: a value type X equipped with an empty instance, an operation to add a natural number to an instance, and an operation to remove a natural number. Which abstract data type is being described, though—stacks, queues, sets, priority queues, or something else?

3.1 Gluing along abstraction functions

To make precise which of these choices the interface PreQueue is meant to classify, we may restrict to inhabitants of this type that have a mandated behavior $q_0 : \bigcirc\textsc{PreQueue}$ as in Section 2.1, creating an algorithmic type. For example, to describe queues, we might choose q_0 to be a cost-free implementation of a queue based on lists:

$$\textsc{Queue} \coloneqq \{\textsc{PreQueue} \mid \P_{\mathsf{beh}} \hookrightarrow (\textsc{List}\ \mathbb{N}, [], (\lambda l.\ \lambda n.\ \mathbf{ret}(l \mathbin{+\!\!+} [n])), \mathit{uncons})\}$$

By restricting the behavior of the given operations to match the intended behavior of a queue, this algorithmic type Queue exactly represents the queue abstract data type. Via the structure identity principle (Eq. (1b)), an element of this type consists of a quadruple

$$\begin{aligned}
(X, p) &: \{\mathcal{V} \mid \P_{\mathsf{beh}} \hookrightarrow \textsc{List}\ \mathbb{N}\} \\
\mathsf{empty} &: \{X \mid \P_{\mathsf{beh}}[p] \hookrightarrow []\} \\
\mathsf{enqueue} &: \{\mathbf{U}(X \rightharpoonup \mathbb{N} \rightharpoonup \mathbf{F}(X)) \mid \P_{\mathsf{beh}}[p] \hookrightarrow \lambda l.\ \lambda n.\ \mathbf{ret}(l \mathbin{+\!\!+} [n])\} \\
\mathsf{dequeue} &: \{\mathbf{U}(X \rightharpoonup \mathbf{F}(\mathbb{N} \times X)) \mid \P_{\mathsf{beh}}[p] \hookrightarrow \mathit{uncons}\}
\end{aligned}$$

$$
\begin{array}{l}
\mathit{batchedQueue} : \textsc{Queue} \\
\mathit{batchedQueue}.X \coloneqq \mathsf{Glue}(\bigcirc(\textsc{List}\ \mathbb{N}), \bullet(\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}), \bullet(\mathit{revAppend}\,;\eta^\circ)) \\
\mathit{batchedQueue}.\mathsf{empty} \coloneqq (\eta^\circ [], \eta^\bullet([], [])) \\
\mathit{batchedQueue}.\mathsf{enqueue}\ (x_\circ, x_\bullet)\ n \coloneqq \\
\quad \mathbf{ret}((\lambda(\_ : \P_{\mathsf{beh}}).\ x_\circ(\_) \mathbin{+\!\!+} [n]), \bullet(\lambda(l_1, l_2).\ (n :: l_1, l_2))(x_\bullet)) \\
\mathit{batchedQueue}.\mathsf{dequeue}\ (x_\circ, \eta^\bullet(l_1, n :: ns)) \coloneqq \mathbf{ret}(n, (\lambda(\_ : \P_{\mathsf{beh}}).\ \mathit{tail}(x_\circ(\_))), \eta^\bullet(l_1, ns)) \\
\mathit{batchedQueue}.\mathsf{dequeue}\ (x_\circ, \eta^\bullet(l_1, []))\ \mathbf{with}\ \mathbf{charge}\langle |l_1| \rangle(\mathit{reverse}\ l_1) \\
\quad \cdots \mid [] \coloneqq \mathbf{ret}(0, \eta^\circ [], \eta^\bullet([], [])) \\
\quad \cdots \mid n :: ns \coloneqq \mathbf{ret}(n, (\lambda(\_ : \P_{\mathsf{beh}}).\ \mathit{tail}(x_\circ(\_))), \eta^\bullet([], ns)) \\
\mathit{batchedQueue}.\mathsf{dequeue}\ (x_\circ, *)\ \mathbf{with}\ \mathit{uncons}(x_\circ(\_)) \\
\quad \cdots \mid (n, l) \coloneqq \mathbf{ret}(n, \eta^\circ l, *)
\end{array}
$$

Fig. 2. The definition of a batched queue, using copattern matching notation [2].

where we write $\{X \mid \P_{\mathsf{beh}}[p] \hookrightarrow x_0(\_)\} \coloneqq \sum_{x:X} \bigcirc(p_*(x) = x_0(\_))$ to classify all $x : X$ that behaviorally match x_0 up to transportation across an equivalence p. Although the behavior of X is fixed to be $\textsc{List}\ \mathbb{N}$, the algorithmic part can be chosen freely so long as it coheres.

Classically, to give a representation type $X_\bullet$ modeled by $\textsc{List}\ \mathbb{N}$, the programmer is to give a meta-theoretic *abstraction function*, $\alpha : X_\bullet \to \textsc{List}\ \mathbb{N}$, that gives the behavioral semantics of an element of $X_\bullet$ as a list.

> The first requirement for the proof [of behavioral correctness] is to define the relationship between the abstract space in which the abstract program is written, and the space of the concrete representation. This can be accomplished by giving a function [α] which maps the concrete variables into the abstract object which they represent… Note that in this and in many other cases [α] will be a many-one function. Thus there is no unique concrete value representing any abstract one. [23]

In this phase-separated setting such abstraction functions become first-class notions, fused into the very definition of the representation type by gluing. To choose a type X with $X \doteq \textsc{List}\ \mathbb{N}$, *i.e.* $\bigcirc X = \bigcirc(\textsc{List}\ \mathbb{N})$, we have by the fracture property (Theorem 1.7) that it suffices to give an algorithmic type $X_\bullet : \mathcal{V}_\bullet$ and a map $\alpha : X_\bullet \to \bullet(\bigcirc(\textsc{List}\ \mathbb{N}))$ and glue them with $\bigcirc(\textsc{List}\ \mathbb{N})$.

Example 3.1 (Batched Queue). To implement functional queues with efficient amortized cost, a pair of lists may be used: incoming data is enqueued to the "inbox" list, and outgoing data is usually dequeued from the "outbox" list, unless it is empty, in which case the "inbox" data is moved to the "outbox" [11, 18, 24, 33]. In order to implement a queue in this way, we use gluing, selecting an algorithmic representation type $X_\bullet$ and an abstraction function $\alpha : X_\bullet \to \bullet(\bigcirc(\textsc{List}\ \mathbb{N}))$:

$$X_\bullet \coloneqq \bullet(\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}) \qquad\qquad \alpha \coloneqq \bullet(\mathit{revAppend}\,;\eta^\circ)$$

where $\mathit{revAppend}\ (l_1, l_2) = l_2 \mathbin{+\!\!+} \mathit{rev}(l_1)$. Encoded within the type $\mathsf{Glue}(\bigcirc(\textsc{List}\ \mathbb{N}), X_\bullet, \alpha)$ itself is the idea that every pair of lists can be transformed to its abstract single-list representation by appending the reversed inbox list to the outbox. Using this gluing as the representation type, we define $\mathit{batchedQueue} : \textsc{Queue}$ in Fig. 2.

The implementation of empty pairs a behavioral empty list and an algorithmic pair of empty lists; the implementation of enqueue incorporates number n into both the behavioral list $x_\circ : \bigcirc(\textsc{List}\ \mathbb{N})$ and the algorithmic pair of lists $x_\bullet : \bullet(\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N})$; and the implementation of dequeue implements batched queue dequeue when $x_\bullet = \eta^\bullet(l_1, l_2)$ and implements the required behavior when $x_\bullet = *$. Within this code, some important proofs are omitted for readability:

(1) Throughout the code, each glued pair $(x_\circ, x_\bullet) : \bigcirc(\textsc{List}\ \mathbb{N}) \times \bullet(\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N})$ comes equipped with a proof that the components cohere according to α.
(2) The cases $\eta^\bullet$ and $*$ must (behaviorally) agree, by the definition of the algorithmic modality.
(3) This definition must come with a proof that each component matches the mandated behavioral specification, q_0. The type matches by the gluing construction (Theorem 1.7), and the operations must be shown to match.

The fact that these definitions behaviorally restrict to q_0 is crucial for implementing the algorithmic type $\textsc{Queue}$: when algorithmic content (cost annotations and the $x_\bullet$ component of $(x_\circ, x_\bullet)$ pairs) is erased in the behavioral phase, we recover the specification queue, q_0, which is the unique behavior of all queues. ⌟

Remark 3.2 (Abstraction Function). The function $\alpha : X_\bullet \to \bullet(\bigcirc(\textsc{List}\ \mathbb{N}))$ being glued along is a linguistic reification of the venerable notion of an *abstraction function* [23], taking a concrete data representation $X_\bullet$ to its representative list. Using the fracture and gluing theorem (Theorem 1.7), we observe that the phase distinction equips every type with an "abstract" component $X_\circ$, a "concrete" component $X_\bullet$, and an abstraction function. Entering the behavioral phase causes all abstraction functions to activate implicitly, leaving behind only "abstract" components and forgetting "concrete" implementation details. This is emphasized through the various semantics of glued type X:

(1) In Semantics 1 the semantics is simply $\textsc{List}\ \mathbb{N}$, retaining the behavioral requirement for the implementation type although obliterating the pair-of-lists split.
(2) In Semantics 2 the semantics recovers the usual functional implementation of batched queues, because the single-list representation is now hidden under an impossible assumption.
(3) In Semantics 3 the semantics maintains both the "concrete" batched and "abstract" single-list representations, as well as the abstraction function *revAppend*, all within type X:

$$[\![X]\!] := [\![\mathsf{Glue}(\bigcirc(\textsc{List}\ \mathbb{N}), \bullet(\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}), \alpha)]\!] = \left(\begin{tikzcd}\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N} \arrow[d, "revAppend"] \\ \textsc{List}\ \mathbb{N}\end{tikzcd}\right)$$

Even the cost model $\mathbb{C}$ has both components: because $\mathbb{C}$ is algorithmic by Axiom 1.9, we have that the "abstract" part of $\mathbb{C}$ is simply trivial. The behavioral phase distinction provides a unifying syntax capable of being compiled to mathematical behavior, efficient algorithms, or both side-by-side. ⌟

Remark 3.3 (Synthetic Parametricity). In their presentation of batched queues Sterling and Harper [43, §4.1] similarly provide a conjoined implementation of a list queue (here, q_0) alongside a batched queue (here, *batchedQueue*) connected by a (functional) relation, *revApp*. Using a pair of symmetric ("left" and "right") phases, either the list queue or the batched queue may be isolated by entering the appropriate phase. Instead, we emphasize here the functional nature of the relation (given as α): we only allow a coercion of our conjoined implementation to the privileged specification q_0, chosen as the canonical meaning of "queue", via the behavioral phase. We recover a theorem analogous to their representation independence result [43, Theorem 4.1] via noninterference (Theorem 1.4): for all result types R and functions $f : \textsc{Queue} \to R$, we have that $f(q_0(\_)) \mathrel{\dot{=}} f(\mathit{batchedQueue})$. ⌟

Remark 3.4 (Relational Correspondence). If two representation types implement the same abstract data type, they can be given a many-to-many heterogeneous relation as is traditional in parametricity arguments [29]. For any types X_1, X_2 equipped with proofs $p_i : X_i \doteq \textsc{List}\ \mathbb{N}$, we can define a relation R as the behavioral pullback of p_1 and p_2:

$$\begin{tikzcd}
 & R \arrow[dl] \arrow[dr] \arrow[dd, phantom, very near start, "\lrcorner" rotate=-45] & \\
X_1 \arrow[d, "\eta^{\circ}_{X_1}"'] & & X_2 \arrow[d, "\eta^{\circ}_{X_2}"] \\
\bigcirc X_1 \arrow[dr, equal, "p_1"'] & & \bigcirc X_2 \arrow[dl, equal, "p_2"] \\
 & \bigcirc(\textsc{List}\ \mathbb{N}) &
\end{tikzcd}
\qquad
R := \sum_{x_1 : X_1} \sum_{x_2 : X_2} p_{1*}(x_1) \doteq_{\textsc{List}\ \mathbb{N}} p_{2*}(x_2)$$

In other words, $x_1 : X_1$ and $x_2 : X_2$ are related exactly when their behaviors as lists match. Note that this type R itself satisfies $R \doteq \textsc{List}\ \mathbb{N}$. Then, queue operations on X_1 and X_2 induce a queue implementation $r : \textsc{Queue}$ with $r.X = R$, recovering an analytic analogue of the synthetic parametricity structures of Sterling and Harper [43] in our synthetic behavioral setting. ⌟

Generalizing this approach beyond queues, we may think of an abstract data type as a behaviorally-fixed type equipped with some behaviorally-fixed operations. To construct an implementation type, we may always use gluing as a canonical technique, because by Theorem 1.7, every type X is constructed via gluing (up to equivalence).

3.2 Compositional verification of behavior

It has long been understood that the behavior of code dependent on abstract data types should only depend on the behavior guaranteed by the ADT.

> If the data representation is proved correct, the correctness of the final concrete program depends only on the correctness of the original abstract program. [23]
>
> When a programmer makes use of an abstract data object, he is concerned only with the behavior which that object exhibits but not with any details of how that behavior is achieved by means of an implementation. [28]

We now capitalize on this principle to verify client programs of the queue abstract data type.

Example 3.5. Consider the following program demonstrating a simple usage pattern for a queue, enqueueing the number 3 to an empty queue and immediately dequeueing:

$$\begin{aligned}
&\mathit{demo} : \mathbf{U}(\textsc{Queue} \rightharpoonup \mathbf{F}(\mathbb{N})) \\
&\mathit{demo}\, q := \mathbf{let}\ \mathbf{ret}(x) = q.\mathrm{enqueue}\ (q.\mathrm{empty})\ 3\ \mathbf{in}\ q.\mathrm{dequeue}\ x
\end{aligned}$$

To verify the behavior of *demo*, we show that $\mathit{demo}\, q \doteq \mathbf{ret}(3)$ for all queue implementations $q : \textsc{Queue}$. As in Section 2.3, we may choose an arbitrary representative implementation, such as *batchedQueue*, without loss of generality. The behavioral equivalence between $\mathit{demo}(\mathit{batchedQueue})$ and $\mathbf{ret}(3)$ holds judgmentally, so the theorem holds. ⌟

Example 3.6. Another simple example usage of queues is to reverse a list by enqueueing its elements and dequeueing them in reverse order, implemented in *qreverse* in Fig. 3. We may verify that the behavior of this *qreverse* function matches the usual list reverse specification, *reverse*: we show that $\mathit{qreverse}\, q \doteq \mathit{reverse}$ for all queue implementations $q : \textsc{Queue}$. Under the behavioral phase, we know by construction that q is equivalent to the specification queue, $q_0(\_)$; in other words, $q \doteq q_0(\_)$. Swapping the queue usages in *fromList* and *toList* for the queue specification $q_0(\_)$, we can see that *fromList* is exactly the *reverse* function, and *toList* is exactly the identity function on lists. Thus, the behavior of *qreverse* is equivalent to *reverse*. Although it is possible to

```
$fromList : \mathbf{U}(\prod_{q:\textsc{Queue}} \textsc{List}\ \mathbb{N} \rightharpoonup \mathbf{F}(q.X))$
$fromList\ q\ []$ := $\mathbf{ret}(q.\mathrm{empty})$
$fromList\ q\ (n :: ns)$ :=
    $\mathbf{let\ ret}(queue) = fromList\ q\ ns\ \mathbf{in}$
    $q.\mathrm{enqueue}\ n\ queue$
```

```
$toList$ :
    $\mathbf{U}(\prod_{q:\textsc{Queue}} \mathbb{N} \rightharpoonup q.X \rightharpoonup \mathbf{F}(\textsc{List}\ \mathbb{N}))$
$toList\ q\ \mathrm{zero}\ x$ := $\mathbf{ret}([])$
$toList\ q\ (\mathrm{suc}\ k)\ x$ :=
    $\mathbf{let\ ret}(n, x') = q.\mathrm{dequeue}\ x\ \mathbf{in}$
    $\mathbf{let\ ret}(ns) = toList\ q\ k\ x'\ \mathbf{in}$
    $\mathbf{ret}(n :: ns)$
```

```
$qreverse$ :
    $\mathbf{U}(\textsc{Queue} \rightharpoonup \textsc{List}\ \mathbb{N} \rightharpoonup \mathbf{F}(\textsc{List}\ \mathbb{N}))$
$qreverse\ q\ l$ :=
    $\mathbf{let\ ret}(queue) = fromList\ q\ l\ \mathbf{in}$
    $toList\ q\ |l|\ queue$
```

```
$reverse : \mathbf{U}(\textsc{List}\ \mathbb{N} \rightharpoonup \mathbf{F}(\textsc{List}\ \mathbb{N}))$
$reverse\ []$ := $\mathbf{ret}([])$
$reverse\ (n :: ns)$ :=
    $\mathbf{let\ ret}(ns') = reverse\ ns\ \mathbf{in}$
    $\mathbf{ret}(ns' \mathbin{+\!\!+} [n])$
```

Fig. 3. List reverse implemented using a queue, *qreverse*, and a direct list reversal function, *reverse*.

verify this fact about *batchedQueue* directly, the proof is more involved; using the algorithmicity of the Queue type allows us to choose a convenient implementation for verification of this fact, such as the list representation of queues, $q_0(\_)$. ⌟

4 BEHAVIORAL PROPERTIES AND DATA STRUCTURE QUOTIENTS

Many common abstract data types classify free/inductive algebraic structures. For example, finite ordered sequences are classified by the free monoid, finite multisets are classified by the free commutative monoid, and finite sets are classified by the free semilattice. In this section using finite ordered sequences as our primary example, we explore how to form an algorithmic type that classifies data structures implementing a free algebraic structure, and we use behavioral quotients to implicitly imbue implementations with abstraction functions.

4.1 Behavioral properties

Behaviorally, it is well understood that the free monoid classifies finite ordered sequences of data, henceforth referred to as *sequences* [9]. Free monoids are unique up to behavioral equivalence: mathematically speaking, lists are the only free monoid. Algorithmically, though, there are many ways to implement data structures modeling the free monoid, all with different cost profiles: arrays, ("linked") lists, finite functions, trees, and various balanced trees, to name a few. Each implementation consists of a type A equipped with a "raw monoid" structure (including the capability for effects):

$$\textsc{RawMonoid} := \sum_{A:C} (\mathrm{empty} : \mathbf{U}(A)) \times (\mathrm{append} : \mathbf{U}(A \otimes A \multimap A))$$

To be considered a monoid, these operations must satisfy some additional properties, identity and associativity, rendered in the presence of effects as follows:

$\textsc{LeftIdentity}(\mathrm{emp}, \mathrm{app}) := \prod_{a:\mathbf{U}(A)} \mathrm{app}(\mathrm{emp}, a) = a$
$\textsc{RightIdentity}(\mathrm{emp}, \mathrm{app}) := \prod_{a:\mathbf{U}(A)} \mathrm{app}(a, \mathrm{emp}) = a$
$\textsc{Associative}(\mathrm{app}) := \prod_{a_1,a_2,a_3:A} \mathrm{app}(\mathrm{app}(a_1, a_2), a_3) = \mathrm{app}(a_1, \mathrm{app}(a_2, a_3))$

As in the Agda standard library[46], we package these definitions as follows:

$\textsc{Identity}(\mathrm{emp}, \mathrm{app}) := \textsc{LeftIdentity}(\mathrm{emp}, \mathrm{app}) \times \textsc{RightIdentity}(\mathrm{emp}, \mathrm{app})$

$$\begin{aligned}
&\mathit{addMonoid} : \textsc{Monoid}^{\circ}\textsc{On}\ \mathbb{N}\\
&\mathit{addMonoid}.A \coloneqq \mathbf{F}(\mathbb{N})\\
&\mathit{addMonoid}.\mathsf{empty} \coloneqq \mathbf{ret}(0)\\
&\mathit{addMonoid}.\mathsf{append}\ (\mathbf{ret}(n_1), \mathbf{ret}(n_2)) \coloneqq \mathbf{charge}\langle 1\rangle(\mathbf{ret}(n_1 + n_2))\\
&\mathit{addMonoid}.\mathsf{singleton}\ n \coloneqq \mathbf{ret}(n)
\end{aligned}$$

Fig. 4. Behavioral monoid implementing addition of natural numbers, instrumented with a cost of 1 per addition operation. The behavioral IsMonoid proof is omitted for brevity.

$$\textsc{IsMonoid}(\mathsf{emp}, \mathsf{app}) \coloneqq \textsc{Associative}(\mathsf{app}) \times \textsc{Identity}(\mathsf{emp}, \mathsf{app})$$

In the presence of cost, though, implementations of empty and append only satisfy these properties behaviorally.

Example 4.1. Lists with a cost model considering recursive calls form a raw monoid:

$$\begin{aligned}
&\mathit{listRawMonoid} : \textsc{RawMonoid}\\
&\mathit{listRawMonoid}.A \coloneqq \mathbf{F}(\textsc{List}\ \mathbb{N})\\
&\mathit{listRawMonoid}.\mathsf{empty} \coloneqq \mathbf{ret}([])\\
&\mathit{listRawMonoid}.\mathsf{append}\ (\mathbf{ret}(l_1), \mathbf{ret}(l_2)) \coloneqq \mathbf{charge}\langle |l_1|\rangle(\mathbf{ret}(l_1 +\!\!+\ l_2))
\end{aligned}$$

However, this raw monoid does not satisfy the axioms for a monoid when cost is under consideration. Although the append function does have empty as a left identity,

$$\mathit{listRawMonoid}.\mathsf{append}\ (\mathbf{ret}([]), \mathbf{ret}(l)) = \mathbf{charge}\langle 0\rangle(\mathbf{ret}([] +\!\!+\ l)) = \mathbf{ret}([] +\!\!+\ l) = \mathbf{ret}(l),$$

it only *behaviorally* has empty as a right identity for non-empty l,

$$\mathit{listRawMonoid}.\mathsf{append}\ (\mathbf{ret}(l), \mathbf{ret}([])) = \mathbf{charge}\langle |l|\rangle(\mathbf{ret}([] +\!\!+\ l)) \doteq \mathbf{ret}([] +\!\!+\ l) = \mathbf{ret}(l),$$

since the left side costs $|l|$ and the right side costs zero. Associativity also holds only behaviorally. ⌟

Since the monoid properties can only be expected to hold behaviorally, we work not with monoids, but instead with *behavioral monoids*:

$$\textsc{Monoid}^{\circ} \coloneqq \sum_{(A,\mathsf{empty},\mathsf{append}):\textsc{RawMonoid}} \bigcirc(\textsc{IsMonoid}(\mathsf{empty}, \mathsf{append}))$$

In this definition a behavioral monoid consists of a raw monoid equipped with a proof that behaviorally, the raw monoid operations indeed form a monoid. Note that behaviorally, a behavioral monoid is just a monoid: entering the behavioral phase recovers the usual notion of a monoid.

Now, to develop sequences, we further equip a behavioral monoid with a generator, serving to create a singleton sequence, for an element type $E : \mathcal{V}$:

$$\textsc{Monoid}^{\circ}\textsc{On}\ E \coloneqq \sum_{M:\textsc{Monoid}^{\circ}} (\mathsf{singleton} : \mathbf{U}(E \rightharpoonup M.A))$$

For example, we can implement a behavioral monoid on $\mathbb{N}$ that performs addition as shown in Fig. 4, even when the addition operation is annotated with cost such that it only satisfies the identity laws in the behavioral phase.

To describe ordered sequences on E, we additionally ask for a mapreduce function for each other behavioral monoid on E, the recursor sending a sequence of type $M.A$ to the carrier of another behavioral monoid, $M'.A$:

$$\textsc{PreSequence}\ E \coloneqq \sum_{M:\textsc{Monoid}^{\circ}\textsc{On}\ E} \left(\mathsf{mapreduce} : \prod_{M':\textsc{Monoid}^{\circ}\textsc{On}\ E} \mathbf{U}(M.A \multimap M'.A)\right)$$

$$\begin{aligned}
&\mathit{listPreSequence}_E : \textsc{PreSequence}\ E\\
&\mathit{listPreSequence}_E.A \coloneqq \mathbf{F}(\textsc{List}\ E)\\
&\mathit{listPreSequence}_E.\mathsf{empty} \coloneqq \mathbf{ret}([\,])\\
&\mathit{listPreSequence}_E.\mathsf{append}\ (\mathbf{ret}(l_1), \mathbf{ret}(l_2)) \coloneqq \mathbf{charge}\langle |l_1| \rangle(\mathbf{ret}(l_1 + l_2))\\
&\mathit{listPreSequence}_E.\mathsf{singleton}\ e \coloneqq \mathbf{ret}([e])\\
&\mathit{listPreSequence}_E.\mathsf{mapreduce}\ M' \coloneqq \mathit{foldr}\ (M'.\mathsf{empty})\ (\lambda e\ a.\ M'.\mathsf{append}\ (M'.\mathsf{singleton}\ e, a))
\end{aligned}$$

Fig. 5. Implementation of PreSequence E using the list type as the representation type.

Example 4.2. We may define implement PreSequence E using the list type as a representative implementation, as shown in Fig. 5. To implement mapreduce, we use the structure of M' to combine the list elements. ⌟

As with the type PreQueue used in the queue example of Section 3, there are many possible implementations of PreSequence E. To adapt PreSequence E into an algorithmic type, we must restrict the behavior of implementations to ensure that the monoid operations and mapreduce are universal. One viable strategy would be to require behavioral coherence with *listPreSequence*$_E$:

$$\textsc{Sequence}\ E \coloneqq \{\textsc{PreSequence}\ E \mid \P_{\mathsf{beh}} \hookrightarrow \mathit{listPreSequence}_E\}$$

However, just as the "unbiased" definition of Sort from Example 2.2 was more ergonomic than the "*isort*-biased" definition from Example 2.1, we may use the universality of the free monoid to provide an equivalent unbiased definition of Sequence E more ergonomic for verification.

4.2 Phased universal properties

To state the universality of the free monoid, we will ask that mapreduce be some form of homomorphism. However, when cost is considered, mapreduce need not preserve the Monoid°On E structure. Thus, we will ask that mapreduce only *behaviorally* preserve structure.

Definition 4.3. Let M, M' : Monoid°On E. A *behavioral homomorphism* from M to M' consists of a function $a : M.A \multimap M'.A$ that behaviorally preserves the operations:

$$\begin{aligned}
a(M.\mathsf{empty}) &\doteq M'.\mathsf{empty}\\
a \circ M.\mathsf{append} &\doteq M'.\mathsf{append} \circ (a \otimes a)\\
a \circ M.\mathsf{singleton} &\doteq M'.\mathsf{singleton}
\end{aligned}$$

We write the type of behavioral homomorphisms as $\mathsf{Hom}^{\bigcirc}(M, M')$.

Using behavioral homomorphisms, we can define the algorithmic type of sequences to be the behaviorally-initial Monoid°On E:

$$\textsc{Sequence}\ E \coloneqq \sum_{M:\textsc{Monoid}^\circ\textsc{On}\ E}\ \prod_{M':\textsc{Monoid}^\circ\textsc{On}\ E}\ \sum_{\alpha:\mathsf{Hom}^{\bigcirc}(M,M')}\ \prod_{\alpha':\mathsf{Hom}^{\bigcirc}(M,M')} \bigcirc(\alpha = \alpha').$$

In other words a sequence consists of M : Monoid°On E with a representation type and empty, append, and singleton constructors, alongside a mapreduce function (taking M') whose universal property is guaranteed by the behavioral uniqueness and preservation of the monoid structure. In the behavioral phase this type restricts to the usual mathematical definition of an initial object of type Monoid°On E, which must be unique; this causes the type Sequence E to be algorithmic.

Example 4.4. Paired with a proof that mapreduce is behaviorally the unique homomorphism from M to M', the data *listPreSequence*$_E$ comprises an implementation *listSequence*$_E$: Sequence E. ⌟

$$
\begin{array}{l}
\textbf{data}\ \textsc{Tree}\ (E : \mathcal{V}) : \mathcal{V}\ \textbf{where} \\
\quad \mathsf{empty} : \textsc{Tree}\ E \\
\quad \mathsf{leaf} : E \to \textsc{Tree}\ E \\
\quad \mathsf{node} : \textsc{Tree}\ E \to \textsc{Tree}\ E \to \textsc{Tree}\ E \\
\quad \mathsf{id}^{\mathsf{l}} : \prod_{t:\textsc{Tree}\ E} \mathsf{node}\ t\ \mathsf{empty} \mathrel{\dot{=}} t \\
\quad \mathsf{id}^{\mathsf{r}} : \prod_{t:\textsc{Tree}\ E} \mathsf{node}\ \mathsf{empty}\ t \mathrel{\dot{=}} t \\
\quad \mathsf{assoc} : \prod_{t_1,t_2,t_3:\textsc{Tree}\ E} \mathsf{node}\ (\mathsf{node}\ t_1\ t_2)\ t_3 \mathrel{\dot{=}} \mathsf{node}\ t_1\ (\mathsf{node}\ t_2\ t_3)
\end{array}
$$

Fig. 6. Type representing binary trees behaviorally quotiented by the monoid laws.

Without loss of generality, then, we may view $\eta^\circ(\mathit{listSequence}_E)$ as the behavior of Sequence E.

4.3 Behavioral quotients as representation types

In Section 3.1 we use gluing to construct a type that is behaviorally equivalent to the given specification. Although this approach is technically always applicable by Theorem 1.7, storing both representations side-by-side is not particularly ergonomic: in the operations provided by an implementation, we must duplicate inline the behavioral operations (using lists) and show that our implementation coheres. Especially when working with an "unbiased" universal construction, such as the behaviorally-free monoid, it is often more straightforward to verify the correctness of an implementation on its own terms.

Remark 4.5. A similar issue of ergonomics occurs in phase distinctions for module systems [22]: it is inconvenient to "physically" separate types and terms in modules, referred to as a *phase separation.* Instead, it is preferable to intermix types and terms and isolate the static type components, using a *phase distinction* as in the module calculus of Sterling and Harper [43]. ⌟

We now present an alternative approach to remove this redundancy: using behavioral quotients (as in Section 2.2), we may manually collapse our representation type to the specification when inside the phase, blending the behavioral reference into the representation type by specifying with modalities and quotients what data to behaviorally erase. Moreover, this collapsing can occur locally, only involving a particular pattern of constructors rather than a recursive definition.

Example 4.6. Sequences may be implemented as trees, causing the constructors empty, append, and singleton to incur zero cost. We construct a behaviorally quotiented type of binary trees that collapses to lists under the phase. To accomplish this, we behaviorally quotient by identity and associativity, as shown in Fig. 6.

The behavioral phase is intended to erase details only included for efficiency. Here, we erase the tree shape in which the elements are stored within the behavioral phase, leaving over only the elements in their given order. Now, when implementing an operation that cases on a tree of type Tree E, we must verify that the code behaviorally respects removal of empty constructors and tree rotations; this rules out functions that behaviorally reveal information about the chosen tree representation of a sequence, even though the choice of representation may algorithmically have an impact on the (cost of the) result. For example, by noninterference (Theorem 1.4), it is impossible to write a function Tree $E \to \mathbb{N}$ that behaviorally computes the height of the input tree. It is important that the quotient only applies in the behavioral phase: if the tree quotient applied more generally, we would lose the ability to write most algorithms, as the cost annotations on an algorithm need not respect the quotients.

Implementing Sequence E, the type component is chosen to be as $\mathbf{F}(\textsc{Tree}\ E)$. This greatly simplifies the implementation: rather than managing a list and a tree side-by-side and ensuring the

$$
\begin{aligned}
&\mathit{treeSequence}_E : \textsc{Sequence}\ E\\
&\mathit{treeSequence}_E.A \coloneqq \mathbf{F}(\textsc{Tree}\ E)\\
&\mathit{treeSequence}_E.\mathsf{empty} \coloneqq \mathbf{ret}(\mathsf{empty})\\
&\mathit{treeSequence}_E.\mathsf{append}\ (\mathbf{ret}(t_1), \mathbf{ret}(t_2)) \coloneqq \mathbf{ret}(\mathsf{node}(t_1, t_2))\\
&\mathit{treeSequence}_E.\mathsf{singleton}\ e \coloneqq \mathbf{ret}(\mathsf{leaf}(e))\\
&\mathit{treeSequence}_E.\mathsf{mapreduce}\ M'\ (\mathbf{ret}(\mathsf{empty})) \coloneqq M'.\mathsf{empty}\\
&\mathit{treeSequence}_E.\mathsf{mapreduce}\ M'\ (\mathbf{ret}(\mathsf{leaf}\ e)) \coloneqq M'.\mathsf{singleton}\ e\\
&\mathit{treeSequence}_E.\mathsf{mapreduce}\ M'\ (\mathbf{ret}(\mathsf{node}(t_1, t_2))) \coloneqq\\
&\quad M'.\mathsf{append}\ (\mathit{treeSequence}_E.\mathsf{mapreduce}\, M'\ (\mathbf{ret}(t_1)), \mathit{treeSequence}_E.\mathsf{mapreduce}\, M'\ (\mathbf{ret}(t_2)))
\end{aligned}
$$

Fig. 7. Implementation of sequences using the binary tree type of Fig. 6.

coherence of operations with in-order traversal explicitly, as gluing would require, we simply operate on trees. The list representation implicitly reveals itself in the behavioral phase due to the quotient cases, which locally and unobtrusively ensures coherence with in-order traversal. We show the implementation in Fig. 7, omitting proofs as usual. The proof that mapreduce respects the behavioral quotient cases follows from the assumption that M' is a behavioral monoid. ⌟

Remark 4.7. In Semantics 3, the type $\textsc{Tree}\ E$ may be interpreted as the presheaf

$$\left(\begin{tikzcd}\textsc{BinaryTree}\ E \arrow[d, "\mathit{inOrder}"] \\ \textsc{List}\ E\end{tikzcd}\right),$$

where $\textsc{BinaryTree}\ E$ is the usual type of binary trees with elements of type E at the leaves and the abstraction function *inOrder* is the in-order traversal function. Notice that this function was never given explicitly in the syntax, as $\textsc{Tree}\ E$ was not constructed syntactically via gluing! However, it appears semantically due to the behavioral quotient laws: as empty trees may be removed and nodes may be freely associated (without loss of generality, to the right), the behavior of every tree is equivalent to a right-spine, which is simply a list. The use of behavioral quotients here, as opposed to gluing, streamlines programming with $\textsc{Tree}\ E$: functions implemented on this type only need to respect the behavioral quotient laws, without need for a duplicated program in the syntax and a proof of coherence by in-order traversal. ⌟

Remark 4.8 (Views). From the perspective of Wadler [49], trees can be seen as a *view* of lists that have improved efficiency [40]. Functions *in* and *out* are given to convert between the representations. It is remarked that the correct notion of equality must be carefully selected in order for these functions to be inverses:

> The correctness of the view depends on the equivalence between the various ways of representing a join list; otherwise, the *in* and *out* functions would not be inverses. [49]

In our cost-aware setting we first observe that if the conversion functions *in* and *out* incur cost, they will not generally be inverses, as the round-trip would incur nonzero cost. Moreover, when considering efficiency as a first-class notion, it is clear that trees and lists should *not* always be equivalent: an algorithm on a list may have differing efficiencies depending on the chosen associativity of its tree representative! The behavioral quotient recovers the appropriate notion of equivalence, identifying trees containing the same data to make $\textsc{Tree}\ E$ and $\textsc{List}\ E$ behaviorally equivalent. Here, the role of *in* is played by $\eta^\circ : \textsc{Tree}\ E \to \bigcirc(\textsc{Tree}\ E) = \bigcirc(\textsc{List}\ E)$, converting a tree to a list by implicitly forgetting the tree structure under the influence of the phase. The

$$
\begin{array}{l}
\textbf{data}\ \text{IRBTree}\ (c : \bullet\textsc{Color})\ (n : \bullet\mathbb{N})\ (E : \mathcal{V}) : \mathcal{V}\ \textbf{where} \\
\quad \mathsf{empty} : \text{IRBTree}\ (\eta^{\bullet}\mathsf{black})\ (\eta^{\bullet}\mathsf{zero})\ E \\
\quad \mathsf{leaf} : E \longrightarrow \text{IRBTree}\ (\eta^{\bullet}\mathsf{black})\ (\eta^{\bullet}\mathsf{zero})\ E \\
\quad \mathsf{red} : \text{IRBTree}\ (\eta^{\bullet}\mathsf{black})\ n\ E \longrightarrow \text{IRBTree}\ (\eta^{\bullet}\mathsf{black})\ n\ E \longrightarrow \text{IRBTree}\ (\eta^{\bullet}\mathsf{red})\ n\ E \\
\quad \mathsf{black} : \text{IRBTree}\ c_1\ n\ E \longrightarrow \text{IRBTree}\ c_2\ n\ E \longrightarrow \text{IRBTree}\ (\eta^{\bullet}\mathsf{black})\ ((\bullet\mathsf{suc})\ n)\ E \\
\quad \mathsf{recolor} : \P_{\mathsf{beh}} \longrightarrow \prod_{t_1,t_2:\text{IRBTree}\ *\ *\ E} \mathsf{red}\ t_1\ t_2 = \mathsf{black}\ t_1\ t_2 \\
\quad \mathsf{id}^{\mathsf{l}} : \P_{\mathsf{beh}} \longrightarrow \prod_{t:\text{IRBTree}\ *\ *\ E} \mathsf{black}\ t\ \mathsf{empty} = t \\
\quad \mathsf{id}^{\mathsf{r}} : \P_{\mathsf{beh}} \longrightarrow \prod_{t:\text{IRBTree}\ *\ *\ E} \mathsf{black}\ \mathsf{empty}\ t = t \\
\quad \mathsf{assoc} : \P_{\mathsf{beh}} \longrightarrow \prod_{t_1,t_2,t_3:\text{IRBTree}\ *\ *\ E} \mathsf{black}\ (\mathsf{black}\ t_1\ t_2)\ t_3 = \mathsf{black}\ t_1\ (\mathsf{black}\ t_2\ t_3) \\
\\
\text{RBTree}\ E \coloneqq \sum_{c:\bullet\textsc{Color}} \sum_{n:\bullet\mathbb{N}} \text{IRBTree}\ c\ n\ E
\end{array}
$$

Fig. 8. Type representing invariant-preserving red-black trees, instrumented with the algorithmic modality and quotients to behaviorally annihilate red-black coloring and tree shape.

inverse, *out*, is immediate in the behavioral phase, since η° is an algorithmic map (*i.e.*, a behavioral equivalence). ⌟

Example 4.9. To improve efficiency of common operations implemented via mapreduce, a sequence may be implemented using a tree data structure equipped with some additional data to main approximate balance, such as a red-black tree [21, 33].[9] We may adapt the previous example to encode the red-black invariants in a tree, taking care to erase behaviorally-irrelevant information.

Following Weirich [50], we may define an indexed inductive type to enforce the red-black invariants, with indices for tree color and black-height. However, with indices of type Color and $\mathbb{N}$, naively grafting on the associativity and identity laws of Example 4.6 would not even well-typed: since red-black trees must be approximately balanced, performing arbitrary tree rotations need not lead to another invariant-satisfying red-black tree! To evade the red-black invariants, the key maneuver is the placement of the color and black-height invariants under the algorithmic modality: that way, in the behavioral phase, we are no longer obliged to maintain the red-black invariants. Then, to avoid the term-level distinction between black and red nodes—which is only maintained for algorithmic efficiency purposes, anyway—we behaviorally identify both colors of nodes, with the constructor recolor. Finally, we may import the quotients of Example 4.6, using black as the default node color, without loss of generality since red nodes may be behaviorally recolored. We show this quotient inductive type in Fig. 8.

Beyond Tree E, we must now verify that code behaviorally respects recoloring. For example, by noninterference (Theorem 1.4), it is impossible to write a function RBTree $E \longrightarrow$ Color that behaviorally computes the color of a given red-black tree. The Sequence E implementation is a straightforward adaptation of Example 4.6 to account for node coloring, sketched in Fig. 9. The subroutine used to implement the append operation,

$$
\mathit{join} : \mathsf{F}(\text{RBTree}\ E) \otimes \mathsf{F}(\text{RBTree}\ E) \multimap \mathsf{F}(\text{RBTree}\ E),
$$

combines two red-black trees in an order-preserving and invariant-maintaining manner [8, 10] (verified in Calf by Li et al. [27]). Importantly, *join* respects the given behavioral quotients, as it only performs rotations and recolorings to produce a balanced tree. ⌟

[9]Commonly, red-black trees store data at the red and black nodes, used to implement efficient binary search. However, we choose to store data at leaves, simplifying our development involving monoids.

$$\begin{aligned}
&\mathit{rbtSequence}_E : \textsc{Sequence}\ E \\
&\mathit{rbtSequence}_E.A \coloneqq \mathbf{F}(\text{RBT}\textsc{ree}\ E) \\
&\mathit{rbtSequence}_E.\mathsf{empty} \coloneqq \mathbf{ret}(\eta^\bullet\mathsf{black}, \eta^\bullet\mathsf{zero}, \mathsf{empty}) \\
&\mathit{rbtSequence}_E.\mathsf{append} \coloneqq \mathit{join} \\
&\mathit{rbtSequence}_E.\mathsf{singleton}\ e \coloneqq \mathbf{ret}(\eta^\bullet\mathsf{black}, \eta^\bullet\mathsf{zero}, \mathsf{leaf}\ e) \\
&\mathit{rbtSequence}_E.\mathsf{mapreduce} \coloneqq \ldots
\end{aligned}$$

Fig. 9. Representative cases of the implementation of sequences using the red-black tree type of Fig. 8, adapting the implementation of Fig. 7 to balanced trees.

Remark 4.10 (Smart Constructors). For the verification of *join*, the essential lemma is that

$$\mathit{join}(\mathbf{ret}(*, *, t_1), \mathbf{ret}(*, *, t_2)) \mathrel{\mathring{=}}_{\text{IRBT}\textsc{ree}\ *\ *\ E} \mathbf{ret}(\mathsf{black}\ t_1\ t_2).$$

In other words: behaviorally, *join* is just the black constructor. For this reason, we justify the terminology that *join* is a *smart constructor*, informally defined to be a constructor that performs some additional computation for the sake of efficiency only. We may treat this observation as a formal definition: a *smart constructor* is a computation behaviorally equivalent to a constructor. ⌟

Remark 4.11. Since the invariants and quotients are both relative to the behavioral phase, we may extract different results using the various semantics. For example,

(1) in Semantics 1 the indices are erased, and the quotients always apply, recovering the mathematical free monoid (equivalent to $\textsc{List}\ E$); and
(2) in Semantics 2 the algorithmic modality on the indices disappears, and the quotient equations are vacuous, recovering the standard red-black tree type and algorithms.

The phase mediates between these semantics: code must respect the possibility of collapsing to lists under the phase, but this may uniformly be deleted in the semantics to recover the true code. ⌟

4.4 Behavior refinements

Although the elimination form for sequences provides the facility to implement any algorithm, there is no guarantee that the implementation will be efficient. We may refine the type $\textsc{Sequence}\ E$ with extra data, exporting additional operations with known behavior but lower cost, using the fact that algorithmic types are closed under dependent sum [38, Example 1.8].

Lemma 4.12. *If $X : \mathcal{V}$ and $Y : X \to \mathcal{V}$ are algorithmic, then $\sum_{x:X} Y(x)$ is algorithmic.*

Letting $X \coloneqq \textsc{Sequence}\ E$, we may define algorithmic type families Y that equip a sequence with additional data.

Example 4.13. Using any $M : \textsc{Sequence}\ E$, computing the length of a given sequence can be done using mapreduce:

$$\mathit{length} \coloneqq M.\mathsf{mapreduce}\ (\mathbf{F}(\mathbb{N}), \mathbf{ret}(0), \mathit{add}, \mathit{const}(\mathbf{ret}(1)))$$

However, for many implementations of the sequence abstraction, this operation will take linear time. We may refine sequences with an additional primitive operation that must behaviorally cohere with the above implementation:

$$\textsc{SequenceExt}(E) \coloneqq \sum_{M:\textsc{Sequence}\ E} (\mathsf{length} : \{M.A \multimap \mathbf{F}(\mathbb{N}) \mid \P_{\mathsf{beh}} \hookrightarrow \mathit{length}\})$$

Even though this length function must behaviorally cohere with *length* (and could always be implemented as exactly *length*), it may also be implemented using a more efficient method, such as storing the length alongside the sequence for constant-time computation. ⌟

This strategy of exporting behaviorally-redundant information in an abstract data type is pervasive, as the algorithms able to be implemented via the elimination form are rarely the most efficient. For example, the queue abstract data type can be thought of as a refinement of lists with optimized operations for appending elements to the end and removing elements from the beginning, and priority queues can be thought of as refining finite multisets with an optimized operation for removing the least element according to some ordering.

5 CONCLUSION

In this paper we have shown how a synthetic phase distinction explains the key foundations for modularity of algorithms and data structures in dependent type theory. While previous developments in Calf heavily emphasize the behavioral (*i.e.*, open) modality, we bring algorithmic (*i.e.*, closed-modal) types and the related fracture theorem to the forefront for the verification of algorithms and data structures. Furthermore, we emphasize the importance of noninterference not just for guaranteed separation of cost and behavior, but for separation of all private algorithmic concerns, of which cost is a paradigmatic example.

5.1 Related work

We now characterize the relationship between our development and prior work, beyond the connections that have been made throughout the text.

5.1.1 Synthetic phase distinctions. This work is fundamentally built upon the general framework for modalities in homotopy type theory developed by Rijke et al. [38], making particular use of the open and closed modalities associated with the proposition $\P_{\mathsf{beh}}$, and set in the world of synthetic phase distinctions, pioneered by Sterling and Harper [43]. Sterling and Harper [45] also apply the techniques of synthetic phase distinctions to the domain of security and information flow, which broadly aligns with the viewpoint that algorithmic data is private and behavioral data is public.

Recent work by Gratzer et al. [15] on abstraction in dependent type theory makes use of phases to selectively reveal the implementation of definitions; within the phase for a particular definition, the corresponding code is revealed. This technique makes use of extension types [36], similar to the type we write $\{X \mid \P_{\mathsf{beh}} \hookrightarrow x_0(\_)\}$ but equipped with a judgmental rather than typal equality. The judgmental equality ensures a degree of faithfulness to the true source code that our story intentionally avoids: although a private implementation should match its public specification, the proof of this fact is rarely judgmental.

As regards the connections to cost analysis and Calf, the principal reference is Niu et al. [31] on which the cost-oriented discussions in the present paper is based. Therein are provided a comprehensive comparison to related work on formalized cost analysis, all of which applies as well to the present setting.

5.1.2 Ghost code. Our use of phases broadly parallels the technique of *ghost code*, where functional, specification-level ghost code is maintained alongside (typically more efficient) “regular” code. Prior accounts of ghost code have described noninterference of ghost code with regular code, erasing ghost code to extract the efficient regular code [14]. Although our presentation supports the extraction of algorithmic code as an external notion, achieved by giving a semantics where $\P_{\mathsf{beh}}$ is the false proposition as described in Semantics 2, the directionality of our phase is dual: internally, we allow erasure of regular (algorithmic) code, leaving behind only behavior. This ensures our opposite variety of noninterference, of regular code with ghost code (Theorem 1.4), which appears here as the essence of modular verification.

5.1.3 Representation independence and univalence. Angiuli et al. [4] tell a similar story for abstract data types and representation types in a univalent setting. For example, in their presentation of batched queues, the pair-of-lists type is quotiented by equivalence under *revAppend*, leading to a type equivalent to the List $\mathbb{N}$ [4, §4.2]; this is similar to the behavioral quotients we considered in Section 4.3, but implementing the queue example of Section 3. Furthermore, they also discuss truncating a cost counter with the writer monad, propositionally truncating the cost model to identify differing costs [4, Example 2.1]. These quotients are precisely what occurs in our development here under the behavioral phase (or in Semantics 1 where $¶_{\mathsf{beh}}$ is true), so we may think of their work as taking place in the behavioral phase, after all algorithmic details have already been redacted. Thus, our story is pleasingly compatible: to recover the ability to extract code prior to redaction, we simply condition the quotients on the behavioral phase.

5.1.4 Realignment and strict glue type. One role of univalence in this work is strictly equating the glue type $\mathsf{Glue}(X_\circ, X_\bullet, \alpha)$ to its behavioral component $X_\circ$ under the behavioral phase, so that the internal representation of an abstract data type can be related with its specifications as in Section 3.1. A similar result can be achieved in a non-univalent setting by considering the *realignment/strictification axiom* [7, 34, 41, 42] that turns a partial isomorphism under a proposition into a strict equality. Then a *strict glue type* [45, 51] can be defined by realigning the Σ type as we have in $\mathsf{Glue}(X_\circ, X_\bullet, \alpha)$.

5.1.5 Verification of data structures using abstraction functions. This work is far from the first to verify data structures using abstraction functions; for example, Nipkow [30] has developed an extensive suite of data structures in Isabelle with verifications based on abstraction functions. Our development with the behavioral phase can be viewed as a synthetic place in which to reconstruct such analytic arguments: because every type contains an abstraction function, the language provides the capability to uniformly apply all the abstraction functions simultaneously, working in a phase where all data is abstract.

5.1.6 Algebraic specification. In the discipline of algebraic specification abstract data types are specified via equational properties on operations [39]. In general, it is not required that all operations are uniquely defined by the equations. However, this strategy is often too conservative: unless the properties exported are complete with respect to the implementation behavior, there will be theorems that the client wishes to prove that are not consequences of the exported properties, violating the principle of modularity (Corollary 1.5). Moreover, from the perspective of the implementer, it will be possible to provide an “incorrect” implementation if the requirements are not strict enough. These issues are only exacerbated in the presence of complex language features, such as effects and higher-order functions. To constitute an algorithmic type, an algebraic specification must be behaviorally fully constrained.

5.2 Future work

Using the behavioral phase for modularity forces the programmer to carefully construct specifications that are algorithmic, and to carefully redact implementations such that the specification is met. This process distills the essence of each algorithm and data structure, explicitly isolating the choices being made for efficiency, as exemplified in our presentation of red-black trees (Example 4.9). As a general next step, we hope to verify additional algorithms and data structures in this style, as well.

5.2.1 Approximation algorithms. The foundational assumption surrounding the use of algorithmic types for specification is that a unique behavioral specification is known for the problem at hand. In many cases this is true; however, some cases have more relaxed notions of correctness, such as approximation algorithms, algorithms with probabilistic correctness guarantees, or algorithms with numerical error tolerances. Inspired by the work of Atkey [5, §4.3.2] on bounding correctness, we hope to provide a synthetic account of approximation algorithms, generalizing the theory presented here about behavioral singletons to behaviorally *bounded* types.

5.2.2 Cost refinements for abstract data types. In this work we primarily treat cost as a private, algorithmic notion, briefly considering the refinement of specifications for algorithms with cost considerations. We anticipate that this story can be expanded further, providing interfaces for abstract data types that incorporate cost for a compositional story about the verification of cost and behavior. The Decalf [20] type theory adds a judgmental notion of inequality to Calf that compares cost; based on the work of Grodin and Harper [19], which uses (lax) homomorphisms to bound the amortized cost of a data structure, we anticipate that the simple, cost-only notion of inequality provided by Decalf can be generalized to describe homomorphisms, inspired by recent developments in simplicial type theory [16, 17, 36].

DATA AVAILABILITY STATEMENT

Building on the work of Niu et al. [31] and Li et al. [27], the definitions and examples presented have been partially mechanized in the Cubical Agda proof assistant [32].

ACKNOWLEDGMENTS

The authors wish to thank Zachary Battleman, Yue Niu, and Jonathan Sterling for fruitful adjacent collaboration that broadly inspired to this research. This material is based upon work supported by the United States Air Force Office of Scientific Research under grant number FA9550-21-0009 (Tristan Nguyen, program manager) and the National Science Foundation under grant number CCF-1901381. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR or NSF.

REFERENCES

[1] 2024. Abstract Data Type. *Wikipedia* (Oct. 2024). https://en.wikipedia.org/w/index.php?title=Abstract_data_type&oldid=125169777

[2] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. *ACM SIGPLAN Notices* 48, 1 (Jan. 2013), 27–38. https://doi.org/10.1145/2480359.2429075

[3] Danel Ahman, Neil Ghani, and Gordon D. Plotkin. 2016. Dependent Types and Fibred Computational Effects. In *Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science)*, Bart Jacobs and Christof Löding (Eds.). Springer, Berlin, Heidelberg, 36–54. https://doi.org/10.1007/978-3-662-49630-5_3

[4] Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. 2021. Internalizing Representation Independence with Univalence. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 12:1–12:30. https://doi.org/10.1145/3434293

[5] Robert Atkey. 2024. Polynomial Time and Dependent Types. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 76:2288–76:2317. https://doi.org/10.1145/3632918

[6] P. N. Benton. 1995. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models. In *Computer Science Logic (Lecture Notes in Computer Science)*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer, Berlin, Heidelberg, 121–135. https://doi.org/10.1007/BFb0022251

[7] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2016. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 62)*, Jean-Marc Talbot and Laurent Regnier (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:17. https://doi.org/10.4230/LIPIcs.CSL.2016.23

[8] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. *ACM Transactions on Parallel Computing* 9, 2 (April 2022), 7:1–7:41. https://doi.org/10.1145/3512769

[9] Guy E. Blelloch. 1992. *NESL: A Nested Data-Parallel Language.* Technical Report. Carnegie Mellon University, USA.

[10] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/2935764.2935768

[11] F. Warren Burton. 1982. An Efficient Functional Implementation of FIFO Queues. *Inform. Process. Lett.* 14, 5 (July 1982), 205–206. https://doi.org/10.1016/0020-0190(82)90015-1

[12] Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. 2009. Enriching an Effect Calculus with Linear Types. In *Computer Science Logic (Lecture Notes in Computer Science)*, Erich Grädel and Reinhard Kahle (Eds.). Springer, Berlin, Heidelberg, 240–254. https://doi.org/10.1007/978-3-642-04027-6_19

[13] Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. 2014. The Enriched Effect Calculus: Syntax and Semantics. *Journal of Logic and Computation* 24, 3 (June 2014), 615–654. https://doi.org/10.1093/logcom/exs025

[14] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. The Spirit of Ghost Code. *Formal Methods in System Design* 48, 3 (June 2016), 152–174. https://doi.org/10.1007/s10703-016-0243-x

[15] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. 2022. Controlling Unfolding in Type Theory. https://doi.org/10.48550/arXiv.2210.05420 arXiv:2210.05420 [cs]

[16] Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz. 2024. Directed Univalence in Simplicial Homotopy Type Theory. arXiv:2407.09146 [cs, math] http://arxiv.org/abs/2407.09146

[17] Daniel Gratzer, Jonathan Weinberger, and Ulrik Buchholtz. 2025. The Yoneda Embedding in Simplicial Type Theory. https://doi.org/10.48550/arXiv.2501.13229 arXiv:2501.13229 [cs]

[18] David Gries. 1989. *The Science of Programming*. Springer New York.

[19] Harrison Grodin and Robert Harper. 2024. Amortized Analysis via Coalgebra. *Electronic Notes in Theoretical Informatics and Computer Science* Volume 4 - Proceedings of MFPS XL (Dec. 2024). https://doi.org/10.46298/entics.14797

[20] Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. 2024. Decalf: A Directed, Effectful Cost-Aware Logical Framework. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 10:273–10:301. https://doi.org/10.1145/3632852

[21] Leo J. Guibas and Robert Sedgewick. 1978. A Dichromatic Framework for Balanced Trees. In *19th Annual Symposium on Foundations of Computer Science (Sfcs 1978)*. 8–21. https://doi.org/10.1109/SFCS.1978.3

[22] Robert Harper, John C. Mitchell, and Eugenio Moggi. 1989. Higher-Order Modules and the Phase Distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. Association for Computing Machinery, New York, NY, USA, 341–354. https://doi.org/10.1145/96709.96744

[23] C. A. R. Hoare. 1972. Proof of Correctness of Data Representations. *Acta Informatica* 1, 4 (Dec. 1972), 271–281. https://doi.org/10.1007/BF00289507

[24] Robert Hood and Robert Melville. 1981. Real-Time Queue Operations in Pure LISP. *Inform. Process. Lett.* 13, 2 (Nov. 1981), 50–54. https://doi.org/10.1016/0020-0190(81)90030-2

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

[26] Paul Blain Levy. 2003. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Springer Netherlands, Dordrecht. https://doi.org/10.1007/978-94-007-0954-6

[27] Runming Li, Harrison Grodin, and Robert Harper. 2023. A Verified Cost Analysis of Joinable Red-Black Trees. https://doi.org/10.48550/arXiv.2309.11056 arXiv:2309.11056 [cs]

[28] Barbara Liskov and Stephen Zilles. 1974. Programming with Abstract Data Types. *SIGPLAN Not.* 9, 4 (March 1974), 50–59. https://doi.org/10.1145/942572.807045

[29] John C. Mitchell. 1986. Representation Independence and Data Abstraction. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '86)*. Association for Computing Machinery, New York, NY, USA, 263–276. https://doi.org/10.1145/512644.512669

[30] Tobias Nipkow (Ed.). 2024. *Functional Data Structures and Algorithms: A Proof Assistant Approach* (september 22, 2024 ed.). https://functional-algorithms-verified.org/

[31] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A Cost-Aware Logical Framework. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 9:1–9:31. https://doi.org/10.1145/3498670

[32] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. Association for Computing Machinery, New York, NY, USA, 1–2. https://doi.org/10.1145/1481861.1481862

[33] Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press.

[34] Ian Orton and Andrew M. Pitts. 2016. Axioms for Modelling Cubical Type Theory in a Topos. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 62)*, Jean-Marc Talbot and Laurent Regnier (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 24:1–24:19. https://doi.org/10.4230/LIPIcs.CSL.2016.24

[35] Pierre-Marie Pédrot and Nicolas Tabareau. 2019. The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 58:1–58:28. https://doi.org/10.1145/3371126

[36] Emily Riehl and Michael Shulman. 2017. A Type Theory for Synthetic ∞-Categories. *Higher Structures* 1, 1 (Dec. 2017), 147–224. https://doi.org/10.21136/HS.2017.06

[37] Egbert Rijke. 2022. *Introduction to Homotopy Type Theory.* https://arxiv.org/abs/2212.11082

[38] Egbert Rijke, Michael Shulman, and Bas Spitters. 2020. Modalities in Homotopy Type Theory. *Logical Methods in Computer Science* Volume 16, Issue 1 (Jan. 2020). https://doi.org/10.23638/LMCS-16(1:2)2020

[39] Donald Sannella and Andrzej Tarlecki. 2012. *Foundations of Algebraic Specification and Formal Software Development.* Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-17336-3

[40] M. R. Sleep and S. Holmström. 1982. A Short Note Concerning Lazy Reduction Rules for Append. *Software: Practice and Experience* 12, 11 (1982), 1082–1084. https://doi.org/10.1002/spe.4380121109

[41] Jonathan Sterling. 2021. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory.* Ph. D. Dissertation. Carnegie Mellon University.

[42] Jonathan Sterling. 2022. Naïve Logical Relations in Synthetic Tait Computability.

[43] Jonathan Sterling and Robert Harper. 2021. Logical Relations as Types: Proof-Relevant Parametricity for Program Modules. *J. ACM* 68, 6 (Oct. 2021), 41:1–41:47. https://doi.org/10.1145/3474834

[44] Jonathan Sterling and Robert Harper. 2021. A Metalanguage for Multi-Phase Modularity. https://icfp21.sigplan.org/details/mlfamilyworkshop-2021-papers/5/A-metalanguage-for-multi-phase-modularity

[45] Jonathan Sterling and Robert Harper. 2022. Sheaf Semantics of Termination-Insensitive Noninterference. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 228)*, Amy P. Felty (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:19. https://doi.org/10.4230/LIPIcs.FSCD.2022.5

[46] The Agda Community. 2024. Agda Standard Library. https://github.com/agda/agda-stdlib

[47] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Univalent Foundations Program.

[48] Matthijs Vákár. 2017. *In Search of Effectful Dependent Types.* http://purl.org/dc/dcmitype/Text. University of Oxford. https://ora.ox.ac.uk/objects/uuid:e91e19b3-7e10-4fda-9433-f23b469e4049

[49] P. Wadler. 1987. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87).* Association for Computing Machinery, New York, NY, USA, 307–313. https://doi.org/10.1145/41625.41653

[50] Stephanie Weirich. 2014. Depending on Types. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14).* Association for Computing Machinery, New York, NY, USA, 241. https://doi.org/10.1145/2628136.2631168

[51] Zhixuan Yang. 2024. *Structure and Language of Higher-Order Algebraic Effects.* Ph. D. Dissertation. Imperial College London. https://yangzhixuan.github.io/pdf/yang-thesis.pdf