Potential Functions as Types

A Synthetic Modal Formulation of Amortized Cost

HARRISON GRODIN, ETHAN CHU, and RUNMING LI, Carnegie Mellon University, USA
JAN HOFFMANN and ROBERT HARPER, Carnegie Mellon University, USA

Amortized analysis can be framed from the physicist's view, amenable to manual verification in dependent type theory using potential functions, and the banker's view, amenable to automated inference in substructural type theory using type-level credit annotations. In this work, we synthesize these perspectives in Calf, a dependent type theory for cost verification. From the physicist's view, we present a fracture and gluing theorem that renders every type as containing a fusion of an abstraction function and a potential function. By construction, every program between two such types must preserve abstraction, to facilitate modularity of behavior, and conserve potential, to facilitate modularity of cost. Incorporating the banker's view, we synthetically construct type operators for credits and debits. We then define Giralf, a graded substructural dependent type theory for programming with credits and debits, which is semantically interpreted as a sub-language of Calf. Finally, we adapt an inference algorithm to transform a limited class of Calf programs into Giralf counterparts, automating the cost analysis of common algorithms in Calf.

Additional Key Words and Phrases: abstract data type, abstraction, abstraction function, amortized analysis, algorithm analysis, call-by-push-value, cost analysis, data structure, dependent type theory, information flow, modal type theory, modularity, phase distinction, proof assistants, resource analysis, verification

1 INTRODUCTION

Amortized analysis, pioneered by Sleator and Tarjan [1985], is a technique for analyzing the cost of a sequence of operations on an ephemeral data structure. Since its inception, there have been two compatible perspectives of the method—the *physicist's view* and the *banker's view*.

In the physicist's view, a *potential function* $\Phi : X \to \mathbb{C}$ assigns potential (*i.e.*, future cost) to each data structure of a type X, where $\mathbb{C}$ is a type representing cost (commonly the natural numbers). Then, for an operation $f : X \to X$ with a *true cost* $c_\top : X \to \mathbb{C}$ and an imagined *amortized cost* $c_{\text{abs}} : X \to \mathbb{C}$, one proves a principle tantamount to the conservation of energy:

$$c_\top(x) + \Phi(f(x)) \leq \Phi(x) + c_{\text{abs}}(x). \tag{1}$$

Iterating this inequality (traditionally via a telescoping sum) ensures that the true cost of a sequence of operations is upper-bounded by the sum of the amortized costs and the initial potential. Because it requires a proof of Eq. (1), which could rely on arbitrarily complex facts and invariants of the data, the physicist's view is well-suited for *manual verification* in dependent type theory [Grodin and Harper 2024; Niu et al. 2022] and higher-order logic [Nipkow and Brinkop 2019].

In the banker's view, cost is viewed as a coin-like resource—called a *credit*—that can be saved within a data structure. Credits can be spent later to offset the cost of an expensive operation; if all true costs are offset by credits, the amortized cost of a sequence of operations is simply the number of credits stored within the input data. Due to their status as a resource, credits must be treated *substructurally*: although credits may be wasted, they may not be duplicated. In many common algorithms and data structures, it is possible to attach the requisite credits to a data structure automatically, placing a credit in exactly the locations where cost will later be incurred. Thus, the banker's view is well-suited for substructural logics and type theories [Atkey 2011; Mével et al. 2019] as well as *automated inference* [Hoffmann and Jost 2022; Hofmann and Jost 2003].

From either perspective, amortization is fundamentally about *modularity*. Amortized analysis does not affect the *implementation* details or the true cost of data structure operations. Instead, it exports a reasonable cost model as a *cost interface* that allows client programs to reason about the cost of operations while encapsulating exactly when costs occur. Both potential functions and

credits are *ghost data*, serving only to mediate between the *private* reality of an implementation and the *public* fictitious amortized costs presented in an interface.

In this work, we develop and implement a technique that unifies the physicist's method and the banker's method in a dependent type theory for modular verification of amortized cost. The physicist's potential functions and associated inequalities are first-class. Every type comes, implicitly or explicitly, equipped with a private potential function; and every program includes, implicitly or explicitly, a proof of the conservation of potential. A phase distinction ensures modularity, providing a stable mathematical model and amortized cost bound against which client programs may be verified. Within this type theory, we define the banker's credits as a type operator. Then, we construct a substructural sub-language for writing programs with credits, and provide an inference algorithm that emits amortized cost upper bounds and corresponding certificates of soundness.

Our approach achieves these goals by synthesizing three main ideas, seamlessly integrating manual verification of amortized cost, modularity via abstraction, and automated cost inference within dependent type theory, depicted in Fig. 1.

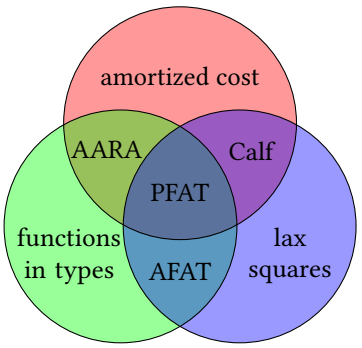


Fig. 1. The central ideas of this work.

(1) We work in Calf [Grodin et al. 2024; Niu et al. 2022], a dependent type theory for cost verification. Within Calf, the conservation of energy principle used in the physicist's view of amortized analysis can be packaged as a lax commutative square [Grodin and Harper 2024].
(2) We make use of the insights of Grodin et al. [2026] who achieve modularity in (univalent) dependent type theory by rendering *abstraction functions as types* (AFAT). Seen via a modal fracture and gluing theorem [Rijke et al. 2020], every type contains an abstraction function, and every function between types contains a commutative square ensuring abstraction is preserved. To accommodate the cost effect of Calf, the authors permit a weaker notion of lax commutativity on costs.
(3) We incorporate ideas from the Automatic Amortized Resource Analysis (AARA) family of substructural type theories [Hoffmann and Jost 2022; Hofmann and Jost 2003], which includes types equipped with credits to represent the banker's view of amortized analysis. Semantically, the types of AARA are interpreted as containing both a set of values and a potential function, and the soundness theorem ensures that potential is conserved. Because credits are spent locally, AARA supports automated inference of cost for common classes of programs, using linear programming to ensure sufficient credits are always available.

Unifying these ideas, we render *potential functions as types* (PFAT). In the following, we provide additional background on each of these ideas, which we then make use of throughout the work.

1.1 Calf: Cost Analysis in Dependent Type Theory

This work takes place in the Calf type theory [Niu et al. 2022], which extends dependent type theory with an adjoint layer supporting a notion of cost. Calf is a dependent variation of call-by-push-value [Ahman et al. 2016; Levy 2003; Pédrot and Tabareau 2019; Vákár 2017]. As such, it includes two sorts of types, the *value types* and the *computation types*:

$$\text{Val. } X, Y, Z ::= \mathbf{U}A \mid 1 \mid X \times Y \mid 0 \mid X + Y \mid \mathbb{N} \mid \textsc{List}\, X \mid \textstyle\sum_{x:X} Y(x) \mid \mathcal{V} \mid C \mid \cdots$$
$$\text{Comp. } A, B, C ::= \mathbf{F}X \mid 1 \mid A \times B \mid X \to A \mid (x : X) \to A(x) \mid \cdots$$

The universe of value types $\mathcal{V}$ and the universe of computation types C are both, themselves, value types [Krishnaswami et al. 2015].

Let $(\mathbb{C}, \leq_{\mathbb{C}}, 0, +)$ be an ordered commutative monoid representing cost, typically chosen to be the natural numbers $(\mathbb{N}, \leq_{\mathbb{N}} 0, +)$. The *cost effect* is a printing-like effect, available at all computation types A: the effect $\mathbf{charge}_A\langle c\rangle(e)$ increases the cost of computation $e : A$ by $c : \mathbb{C}$ units of cost.

$$\frac{\Gamma \vdash c : \mathbb{C} \qquad \Gamma \vdash e : A}{\Gamma \vdash \mathbf{charge}_A\langle c\rangle(e) : A}$$

$$\mathbf{charge}\langle 0\rangle(e) = e$$
$$\mathbf{charge}\langle c_1 + c_2\rangle(e) = \mathbf{charge}\langle c_1\rangle(\mathbf{charge}\langle c_2\rangle(e))$$

Every type A is equipped with a cost preorder $\leq_A$ on UA,[1] where $e \leq_A e'$ means that e and e' have the same behavior, although the cost of e may be lower than that of e' [Grodin et al. 2024].

Within Calf, Niu et al. [2022] verify amortized costs using the physicist's method by proving the requisite conservation principles. Later, in a refinement of Calf, Grodin and Harper [2024] showed that amortized analysis can be viewed as a lax commutative square using additional type constructors of the enriched effect calculus (EEC) [Egger et al. 2009, 2014] and linear/non-linear type theory (LNL) [Benton 1995; Krishnaswami et al. 2015][2], including pure functions $X \to Y$, homomorphisms $A \multimap B$, sums 0 and $A + B$, and copowers $X \pitchfork A$.

$$\text{Val. } X, Y, Z ::= \cdots \mid X \to Y \mid (x : X) \to Y(x) \mid A \multimap B$$
$$\text{Comp. } A, B, C ::= \cdots \mid \underbrace{0 \mid A + B \mid X \pitchfork A \mid (x : X) \pitchfork A(x)}_{\text{EEC}} \mid \underbrace{\top \mid A \otimes B \mid A \multimap B}_{\text{LNL}}$$

Using homomorphisms $A \multimap B$, Grodin and Harper [2024] describe a generalization of amortized analysis centered on computation types. For an implementation type $A_\top$, let $g_\top : A_\top \multimap A_\top$ be an operation on the data structure annotated with a realistic cost model; and for a specification type A_{abs}, let $g_{\text{abs}} : A_{\text{abs}} \multimap A_{\text{abs}}$ be an analogous operation instead annotated with a purported amortized cost. These two programs can be connected by defining a homomorphism $\alpha : A_\top \multimap A_{\text{abs}}$ such that the inequality $\alpha \circ g_\top \leq g_{\text{abs}} \circ \alpha$ holds, depicted as the following lax commutative square:

$$\begin{array}{ccc} A_\top & \xrightarrow{g_\top} & A_\top \\ {\scriptstyle\alpha}\downarrow & \geq & \downarrow{\scriptstyle\alpha} \\ A_{\text{abs}} & \xrightarrow[g_{\text{abs}}]{} & A_{\text{abs}} \end{array}$$

In the case where $A_\top = \mathbf{F}X_\top$ and $A_{\text{abs}} = \mathbf{F}X_{\text{abs}}$, the maps above are of the following form:

$$g_\top\ (\mathbf{ret}\ x_\top) \coloneqq \mathbf{charge}\langle c_\top(x_\top)\rangle(\mathbf{ret}\ (f_\top(x_\top))) \qquad c_\top : X_\top \to \mathbb{C} \qquad f_\top : X_\top \to X_\top$$
$$g_{\text{abs}}\ (\mathbf{ret}\ x_{\text{abs}}) \coloneqq \mathbf{charge}\langle c_{\text{abs}}(x_{\text{abs}})\rangle(\mathbf{ret}\ (f_{\text{abs}}(x_{\text{abs}}))) \qquad c_{\text{abs}} : X_{\text{abs}} \to \mathbb{C} \qquad f_{\text{abs}} : X_{\text{abs}} \to X_{\text{abs}}$$
$$\alpha\ (\mathbf{ret}\ x_\top) \coloneqq \mathbf{charge}\langle \Phi(x_\top)\rangle(\mathbf{ret}\ (\chi(x_\top))) \qquad \Phi : X_\top \to \mathbb{C} \qquad \chi : X_\top \to X_{\text{abs}}$$

Grodin and Harper observe that the cost aspect of the inequality $\alpha \circ g_\top \leq g_{\text{abs}} \circ \alpha$, in this case, is precisely the conservation of potential condition of Eq. (1). For this reason, they refer to the lax commutative square as a "generalized amortization condition" and to α as a "behavior-relevant generalization of potential functions". In this work, inspired by the induced equation $\chi \circ f_\top = f_{\text{abs}} \circ \chi$, we take a dual perspective: we treat α as a cost-aware generalization of an *abstraction function*.

1.2 Abstraction Functions as Types

To support modular abstract data types within dependent type theory, we build on recent work on incorporating Hoare-style *abstraction functions* [Hoare 1972] in dependent type theory [Grodin et al.

[1] Formally, Grodin et al. use a *synthetic* notion of preorder. In univalent type theory, we inherit all types as value types, placing the requirement for a type to be a synthetic preorder on computation types.
[2] The $A \otimes B$ and $A \multimap B$ types of LNL make use the commutativity of the cost monoid.

2026]. The authors propose an *abstract phase*, a proposition **abs**, to isolate abstract/public/interface-level data from concrete/private/implementation-level data using modalities in homotopy type theory [Rijke et al. 2020]. This phase gives rise to a *fracture and gluing* theorem guaranteeing that every type X contains precisely a concrete type, an abstract type, and an abstraction function between the two, all accessible via modal constructions defined using the **abs** phase.

When a function $\chi : X_\top \to X_{\mathrm{abs}}$ is assembled into a type X, functions on X must respect abstraction. For example, a function $f : X \to X$ can be built out of a function $f_\top : X_\top \to X_\top$ on the concrete representation, a function $f_{\mathrm{abs}} : X_{\mathrm{abs}} \to X_{\mathrm{abs}}$ on the abstract representation, and a proof of coherence up to χ (below, left). This technique facilitates modularity by ensuring that, in addition to a concrete implementation, every program also contains a stable abstract specification on which clients can depend. Client code is then verified in the abstract phase (*i.e.*, assuming **abs**), where both the concrete implementation and the abstraction function are erased: $X = X_{\mathrm{abs}}$ and $f = f_{\mathrm{abs}}$.

$$\begin{array}{ccc} X_\top & \xrightarrow{f_\top} & X_\top \\ {\scriptstyle\chi}\downarrow & = & \downarrow{\scriptstyle\chi} \\ X_{\mathrm{abs}} & \xrightarrow[f_{\mathrm{abs}}]{} & X_{\mathrm{abs}} \end{array} \qquad \begin{array}{ccc} X_\top & \xrightarrow{f_\top} & \mathbf{U}(\mathbf{F}X_\top) \\ {\scriptstyle\chi}\downarrow & \geq & \downarrow{\scriptstyle\mathbf{U}(\mathbf{F}\chi)} \\ X_{\mathrm{abs}} & \xrightarrow[f_{\mathrm{abs}}]{} & \mathbf{U}(\mathbf{F}X_{\mathrm{abs}}) \end{array}$$

Grodin et al. [2026] extend this technique to accommodate the cost effect. Let $f_\top$ and f_{abs} be functions satisfying the depicted program inequality (above, right), meaning that they cohere behaviorally and the cost of $f_\top$ is upper-bounded by the cost of f_{abs}. Just as the behavior of f_{abs} is a client-facing approximation of the true behavior of $f_\top$, the cost annotation within f_{abs} is a client-facing upper-bound on the cost of $f_\top$. This construction does not account for amortization exactly because the abstraction function $\chi : X_\top \to X_{\mathrm{abs}}$ is pure and thus cannot incorporate potential via the cost effect. In the present work, we extend this development to enable the assembly of a cost-aware homomorphism $\alpha : A_\top \multimap A_{\mathrm{abs}}$ into a type, thus accommodating both abstraction and potential. The cost of the client-facing $f_{\mathrm{abs}} : A_{\mathrm{abs}} \multimap A_{\mathrm{abs}}$ is then an *amortized* upper-bound on the cost of $f_\top : A_\top \multimap A_\top$, facilitating modular verification with amortized costs.

1.3 AARA: Automatic Amortized Resource Analysis

To automatically infer cost bounds within our type theory, we build on ideas from AARA, a family of substructural type systems [Hofmann and Jost 2003] based on the banker's view. There are many variants of AARA that have been developed over more than two decades [Hoffmann and Jost 2022]; we focus on a core language approximately based on that of Hoffmann and Hofmann [2010b].

Syntactically in AARA, types include credits; for example, the type $\triangleright^c A$ stores c credits alongside data of type A. Semantically, connecting to the physicist's method, every type describes not only a set of values $[\![A]\!]$ but also a potential function $\Phi_A : [\![A]\!] \to \mathbb{C}$ that computes the potential/credits contained within a type. For example, $[\![\triangleright^c A]\!] := [\![A]\!]$ and $\Phi_{\triangleright^c A}(a) := c + \Phi_A(a)$. Syntactically, the affine treatment of credits in AARA ensures that credits are never duplicated; semantically, this appears in the soundness theorem for AARA as a conservation of potential theorem for Φ_A.

The design of AARA is motivated by the desire to reduce automated cost inference to a linear programming (LP) problem. However, this limits expressivity, requiring potential functions (usually polynomials) to be selected *a priori* with limited support for manual verification [Pham et al. 2025].

In reference to languages in the tradition of AARA, Niu et al. make the following remark:

> …it is not immediately clear how one may take better advantage of the existing type-based approaches to amortized analysis in Calf. Niu et al. [2022]

In this work, using the idea that potential functions can be built into types as a bridge between Calf and AARA, we incorporate credits into types to make precisely this connection.

1.4 Contributions

In this work, we synthesize these three major ideas: potential as the cost of an abstraction function, abstraction functions built into types, and credits as a substructural type former.

(1) We extend Calf to natively support a modular account of amortized analysis within dependent type theory, reconstructing the physicist's method. Specifically, in Section 2 we extend the work of Grodin et al. [2026] on building abstraction functions into types, proving a fracture and gluing theorem for the universe of computation types that enable cost-aware abstraction functions that emit potential to be built into types. In Section 3, we observe that this approach facilitates a modular notion of amortized cost interface via the abstract phase.
(2) In Section 4, we develop a standard library for credits, debits, and credit-carrying data structures based on the banker's method inside the dependent type theory.
(3) In Section 5 we define Giralf, an AARA-like graded substructural type theory that streamlines programming with credits and debits. Then, by providing it with a semantics in Calf, we demonstrate formally the sense in which Giralf generalizes AARA. Moreover, we adapt the LP-based cost inference algorithm for AARA to Giralf, generating certificates in Calf guaranteeing the soundness of inferred bounds.

The central theorems and constructions of this work are mechanized in Cubical Agda [Norell 2009; Vezzosi et al. 2019], indicated by the 🤚 symbol.

2 THE PHYSICIST'S VIEW

To incorporate potential functions into types, we extend the technique of Grodin et al. [2026] that builds functions $X_\top \to X_{\mathsf{abs}}$ into value types to the level of computation types, where a homomorphism $A_\top \multimap A_{\mathsf{abs}}$ can emit cost to be thought of as potential. This development takes place in a univalent type theory that is merely extended with a single proposition, the abstract phase **abs**; the rest of the constructions are defined in terms of this proposition.

2.1 Abstraction Functions as Types: Hoare's Abstraction Functions, Synthetically

First, we briefly review the work of Grodin et al. [2026] that makes use of the modalities of Rijke et al. [2020] to incorporate abstraction functions into types. The key result is a *fracture and gluing* theorem stating that every type contains exactly a concrete type, an abstract type, and an abstraction function between them. We first recall the modalities that isolate concrete and abstract types.

Definition 2.1 (Abstract Modality, Grodin et al. 2026 🤚). The *abstract modality* $\bigcirc X \coloneqq \mathbf{abs} \to X$ marks a type as client-facing, where its unit $\eta^\circ_X : X \to \bigcirc X$ is $\lambda\, x \to \lambda\, (\_ : \mathbf{abs}) \to x$. A value type X is *abstract* when η°_X is an equivalence, and write $\mathcal{V}_\circ$ for the universe of abstract value types.

Definition 2.2 (Concrete Modality, Grodin et al. 2026 🤚). The *concrete modality* $\bullet X$ marks a type as irrelevant to clients. It is defined as the following higher-inductive type (a pushout):

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

$$\begin{array}{ccc} \mathbf{abs} \times X & \xrightarrow{\mathrm{proj}_1} & \mathbf{abs} \\ {\scriptstyle \mathrm{proj}_2}\downarrow & & \downarrow{\scriptstyle *} \\ X & \xrightarrow{\eta^\bullet} & \bullet X \end{array}$$

Say that a value type X is *concrete* when $\eta^\bullet_X$ is an equivalence (or equivalently when $\bigcirc X = 1$) and write $\mathcal{V}_\bullet$ for the universe of concrete value types.

For a concrete type $X_\bullet : \mathcal{V}_\bullet$ and an abstract type $X_\circ : \mathcal{V}_\circ$, an *abstraction function* in the style of Hoare [1972] is a function $\chi_\bullet : X_\bullet \to \bullet X_\circ$, where the concrete modality on the output ensures that the abstraction itself is entirely concrete and hidden from the interface. Based on these definitions,

we recall the fracture and gluing theorem, which proves that every type in the universe $\mathcal{V}$ carries precisely the same data as a concrete type, an abstract type, and an abstraction function.

Theorem 2.3 (Fracture and Gluing, Rijke et al. 2020 ✋). *The following equivalence holds:*

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

Proof Sketch. In the forward direction, fracture a type X into the concrete type $X_\bullet := \bullet X$, the abstract type $X_\circ := \circ X$, and the abstraction function $\bullet \eta^\circ_X : \bullet X \to \bullet \circ X$. In the reverse direction, define $\mathbf{Glue}(X_\bullet, X_\circ, \chi_\bullet) := X_\bullet \times_{\bullet X_\circ} X_\circ := \sum_{(x_\bullet, x_\circ) : X_\bullet \times X_\circ} \chi_\bullet(x_\bullet) = \eta^\bullet(x_\circ)$. □

Corollary 2.4 (✋). *Every $f : X \to Y$ consists of a concrete function $f_\bullet : \bullet X \to \bullet Y$ and an abstract function $f_\circ : \circ X \to \circ Y$ that cohere (formally, $\bullet \eta^\circ_Y \circ f_\bullet = \bullet(f_\circ \circ \eta^\circ_X)$).*

Using gluing, it is possible to treat any function $\chi : X_\top \to X_{\text{abs}}$ as an abstraction function $\bullet(\eta^\circ \circ \chi) : \bullet X_\top \to \bullet \circ X_{\text{abs}}$ and thus build it into a type; accordingly, we sometimes refer to χ as an abstraction function. For convenience, we define a utility that builds an arbitrary χ into a type.

Definition 2.5 (✋). Define $\mathbf{Abstraction}(\chi : X_\top \to X_{\text{abs}}) := \mathbf{Glue}(\bullet X_\top, \circ X_{\text{abs}}, \bullet(\eta^\circ \circ \chi))$.

Corollary 2.6 (✋). *By Theorem 2.3, it is the case that $X = \mathbf{Abstraction}(\text{id}_X : X \to X)$ for all X.*

Lemma 2.7 (✋). *To define a map $f : \mathbf{Abstraction}(\chi : X_\top \to X_{\text{abs}}) \to \mathbf{Abstraction}(\psi : Y_\top \to Y_{\text{abs}})$, it suffices by Corollary 2.4 to define a pair of functions $f_\top : X_\top \to Y_\top$ and $f_{\text{abs}} : X_{\text{abs}} \to Y_{\text{abs}}$ such that*

$$\begin{array}{ccc} X_\top & \xrightarrow{f_\top} & Y_\top \\ {\scriptstyle\chi}\downarrow & = & \downarrow{\scriptstyle\psi} \\ X_{\text{abs}} & \xrightarrow[f_{\text{abs}}]{} & Y_{\text{abs}} \end{array}$$

using $f_\bullet = \bullet f_\top$ and $f_\circ = \circ f_{\text{abs}}$.

The equation, rendered as a square, is a functional analogue of the relational notion of representation independence [Reynolds 1983].

Example 2.8 (Grodin et al. 2026, §2.2.1). Consider the batched queue functional data structure, which represents a queue as a pair of lists (l_1, l_2) [Burton 1982; Gries 1989; Hood and Melville 1981; Okasaki 1999]. Elements are enqueued to the "back" list l_2 and dequeued from the "front" list l_1—unless, of course, the second list is empty, in which case the dequeue operation reverses l_2 to replace l_1. Although the implementation type of batched queues is $\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}$, the generic client-facing specification type describing the mathematical behavior of queues is $\textsc{List}\ \mathbb{N}$; the former can be converted to the latter using the function $\chi := \lambda\ (l_1, l_2) \to l_1 +\!\!+ \mathit{reverse}\ l_2$. Both perspectives, mediated by χ, can be built into a single type:

$$X := \mathbf{Abstraction}(\chi : \textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N} \to \textsc{List}\ \mathbb{N}).$$

To define a function $\mathit{enqueue} : \mathbb{N} \to X \to X$, it suffices by Lemma 2.7 to give the true code alongside an abstract mathematical model that coheres up to χ. Defining

$$\mathit{enqueue}_\top\ n\ (l_1, l_2) := (l_1, n :: l_2) \qquad\qquad \mathit{enqueue}_{\text{abs}}\ n\ l := l +\!\!+ [n],$$

observe that the following equation holds:

$$\begin{array}{ccc} \textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N} & \xrightarrow{\mathit{enqueue}_\top\ n} & \textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N} \\ {\scriptstyle\chi}\downarrow & = & \downarrow{\scriptstyle\chi} \\ \textsc{List}\ \mathbb{N} & \xrightarrow[\mathit{enqueue}_{\text{abs}}\ n]{} & \textsc{List}\ \mathbb{N} \end{array}$$

These data suffice to implement *enqueue*, hiding the private batched implementation under a public list-based specification. The *empty* and *dequeue* operations are similar. ⌟

By Theorem 2.3, a programmer knows that every type contains a concrete type $X_\bullet$, an abstract type $X_\circ$, and an abstraction function $\chi_\bullet : X_\bullet \to \bullet X_\circ$. However, the meanings of "concrete" and "abstract" depend on the semantic interpretation of the phase proposition **abs** [Grodin et al. 2026].

Semantics. Using a Kripke (*i.e.*, presheaf) semantics with two worlds $\{\mathsf{abs} \vdash \top\}$, every type is interpreted as a pair of types along with a genuine function, and every function $f : X \to Y$ is interpreted as a coherent pair of functions:

$$[\![X]\!] := \begin{pmatrix} [\![X]\!]_\top \\ \downarrow{\scriptstyle [\![X]\!]_\vdash} \\ [\![X]\!]_{\mathsf{abs}} \end{pmatrix} \qquad [\![f]\!] := \begin{array}{ccc} [\![X]\!]_\top & \xrightarrow{[\![f]\!]_\top} & [\![Y]\!]_\top \\ {\scriptstyle [\![X]\!]_\vdash}\downarrow & = & \downarrow{\scriptstyle [\![Y]\!]_\vdash} \\ [\![X]\!]_{\mathsf{abs}} & \xrightarrow[[\![f]\!]_{\mathsf{abs}}]{} & [\![Y]\!]_\vdash \end{array}$$

2.2 Potential Functions as Types: Sleator's Potential Functions, Synthetically

In the previous section, we recalled that a function $\chi : X_\top \to X_{\mathsf{abs}}$ can be assembled into a value type. Now, we extend this construction to the level of computation types, enabling a homomorphism $\alpha : A_\top \multimap A_{\mathsf{abs}}$ to be assembled into a computation type. Viewing the cost of such homomorphisms as potential à la Grodin and Harper [2024], this will render *potential functions as types*.

With the goal of proving an analogous fracture and gluing principle for the universe $\mathcal{C}$ of computation types, we must first define analogues to the abstract and concrete modalities (Definitions 2.1 and 2.2) at the level of computation types. First, the abstract modality adapts straightforwardly:

Definition 2.9 (✋). The *abstract modality on computation types* is defined as $\bigcirc A := \mathbf{abs} \multimap A$, where its unit $\eta^\circ_A : A \multimap \bigcirc A$ is defined as $\lambda\, a \multimap \lambda\, (\_ : \mathbf{abs}) \multimap a$. Say that a computation type A is *abstract* when η°_A is an equivalence, and write $\mathcal{C}_\circ$ for the universe of abstract computation types.

In Definition 2.2, the concrete modality $\bullet X$ on value types is defined as a pushout of X and **abs** over the product type $\mathbf{abs} \times X$. To define the concrete modality $\bullet A$ on computation types, we replace **abs** and $\mathbf{abs} \times X$ with the copowers $\mathbf{abs} \pitchfork 1$ and $\mathbf{abs} \pitchfork A$, respectively.

Definition 2.10 (✋). The *concrete modality on computation types* is the following pushout:

$$\begin{array}{ccc} \mathbf{abs} \pitchfork A & \xrightarrow{\mathbf{abs} \pitchfork \mathsf{triv}} & \mathbf{abs} \pitchfork 1 \\ {\scriptstyle \mathsf{proj}}\downarrow & & \downarrow{\scriptstyle *} \\ A & \xrightarrow{\eta^\bullet_A} & \bullet A \end{array}$$

Crucially, given $\_ : \mathbf{abs}$, we have $\bullet A = 1$. Say that a computation type A is *concrete* when $\eta^\bullet_A$ is an equivalence, and write $\mathcal{C}_\bullet$ for the universe of concrete computation types.

Although the abstract modality $\bigcirc A$ is well-behaved, the concrete modality $\bullet A$ given above is insufficient[3] for fracture and gluing without further information about the interpretation of computation types. Thus, it is important to reveal additional information about the structure of $\mathcal{C}$.

2.2.1 Computation Types as Cost Algebras. In prior work on Calf, computation types are syntactically left unspecified but semantically interpreted as *cost algebras* [Li and Harper 2025; Niu et al. 2022]. In this work, rather than leaving $\mathcal{C}$ unknown, we *define* $\mathcal{C}$ to be the universe of cost algebras.

[3]This modal operator need not constitute a stable orthogonal factorization system [Rijke et al. 2020].

Definition 2.11 (✋). A *cost algebra* $A : C$ consists of

(1) an underlying preordered[4] value type $\mathbf{U}A : \mathcal{V}$ and
(2) a function $\mathbf{charge}_A\langle - \rangle : \mathbb{C} \to \mathbf{U}A \to \mathbf{U}A$ such that
(3) $\mathbf{charge}_A\langle 0 \rangle(a) = a$ and
(4) $\mathbf{charge}_A\langle c_1 + c_2 \rangle(a) = \mathbf{charge}_A\langle c_1 \rangle(\mathbf{charge}_A\langle c_2 \rangle(a))$.

In other words, a cost algebra consists of a value type along with a sensible way of incorporating cost, used to give the semantics of the cost effect as given above.[5] Correspondingly, we define the value type $A \multimap B$ to be the type of cost algebra homomorphisms.

Definition 2.12 (✋). A *cost algebra homomorphism* $f : A \multimap B$ consists of

(1) an underlying value-level function $\mathbf{U}f : \mathbf{U}A \to \mathbf{U}B$ such that
(2) for all $a : \mathbf{U}A$, it is the case that $\mathbf{charge}_B\langle c \rangle(\mathbf{U}f(a)) = \mathbf{U}f(\mathbf{charge}_A\langle c \rangle(a))$.

For readability, we often write f instead of $\mathbf{U}f$, such as $f(a)$ instead of $\mathbf{U}f(a)$.

Even though such an $f : A \multimap B$ is a reusable value, the notation is inspired by the idea that such a function should use its input linearly, so as not to drop or duplicate incoming effects [Egger et al. 2009, 2014]. This notion of linearity is *semantic*, requiring a proof that cost is preserved.

Given this revelation of computation types as cost algebras, the computation-level abstract and concrete modalities satisfy properties sufficient for proving fracture and gluing.

Lemma 2.13 (✋). *For all computation types A, both $\mathbf{U}(\bigcirc A) = \bigcirc(\mathbf{U}A)$ and $\mathbf{U}(\CIRCLE A) = \CIRCLE(\mathbf{U}A)$.*

Lemma 2.14 (✋). *Both $\bigcirc$ and $\CIRCLE$ are* lex*, meaning that they preserve pullbacks.*

2.2.2 Fracture and Gluing. We now lift the fracture and gluing theorem on the universe of value types $\mathcal{V}$ (Theorem 2.3) to a fracture and gluing theorem on the universe of computation types C, which is revealed to be the universe of cost algebras.

For a concrete type $A_\bullet : C_\bullet$ and an abstract type $A_\circ : C_\circ$, an *abstraction homomorphism* is a homomorphism $\alpha_\bullet : A_\bullet \multimap \CIRCLE A_\circ$. Beyond the abstraction capabilities of an abstraction function $\chi_\bullet : X_\bullet \to \CIRCLE X_\circ$, an abstraction homomorphism may perform the cost effect, which serves the role of potential. In this way, the following theorem explains why every computation type $A : C$ contains not only an abstraction function but also a potential function.

Theorem 2.15 (Fracture and Gluing of C, ✋). *Every computation type $A : C$ contains exactly the data of a concrete type $A_\bullet$, an abstract type $A_\circ$, and an abstraction homomorphism between them:*

$$C = \sum_{A_\bullet : C_\bullet} \sum_{A_\circ : C_\circ} A_\bullet \multimap \CIRCLE A_\circ.$$

Proof Sketch. Fracture a type A into $(\CIRCLE A, \bigcirc A, \CIRCLE \eta^\circ)$, and glue $(A_\bullet, A_\circ, \alpha_\bullet)$ into the pullback

$$\mathbf{Glue}(A_\bullet, A_\circ, \alpha_\bullet) \coloneqq A_\bullet \times_{\CIRCLE A_\circ} A_\circ,$$

following Rijke et al. [2020]. To show that $A = \mathbf{Glue}(\CIRCLE A, \bigcirc A, \CIRCLE \eta^\circ)$, using univalence, it suffices to define a map $f : A \multimap \mathbf{Glue}(\CIRCLE A, \bigcirc A, \CIRCLE \eta^\circ)$ and prove that it is an equivalence. Let

$$f : A \multimap \mathbf{Glue}(\CIRCLE A, \bigcirc A, \CIRCLE \eta^\circ)$$
$$f\ a \coloneqq (\eta^\bullet a, \eta^\circ a)$$

[4] In this setting, a preordered value type is a synthetic preorder [Grodin et al. 2024] that, to accommodate the higher type theory of this work, is an h-set [The Univalent Foundations Program 2013]. Both of these requirements are orthogonality conditions, rendering synthetic preorders as a reflective subuniverse [Rijke et al. 2020] of $\mathcal{V}$.

[5] In follow-up work by Grodin et al. [2024], the semantics of computation types was generalized to support other effects; for simplicity, we only consider cost in this work, although the arguments naturally adapt to the setting of various other effects.

using the modal units $\eta^\bullet : A \multimap \bullet A$ and $\eta^\circ : A \multimap \circ A$. To show f is an equivalence, it suffices to show $\mathbf{U}f : \mathbf{U}A \to \mathbf{U}(\mathbf{Glue}(\bullet A, \circ A, \bullet\eta^\circ))$ is an equivalence because $\mathbf{U} : \mathcal{C} \to \mathcal{V}$ is *conservative*. As

$$\begin{aligned}
\mathbf{U}(\mathbf{Glue}(\bullet A, \circ A, \bullet\eta^\circ)) &= \mathbf{U}(\bullet A \times_{\bullet\circ A} \circ A) \\
&= \mathbf{U}(\bullet A) \times_{\mathbf{U}(\bullet\circ A)} \mathbf{U}(\circ A) && \text{(right adjoints preserve limits)} \\
&= \bullet(\mathbf{U}A) \times_{\bullet\circ(\mathbf{U}A)} \circ(\mathbf{U}A) && \text{(Lemma 2.13)}
\end{aligned}$$

renders $\mathbf{U}f$ as the fracture-and-gluing of the value type $\mathbf{U}A$, this map is an equivalence by the fracture and gluing theorem for $\mathcal{V}$ (Theorem 2.3). In the other direction, we have that

$$\begin{aligned}
&\bullet(\mathbf{Glue}(A_\bullet, A_\circ, \bullet\eta^\circ)) && \circ(\mathbf{Glue}(A_\bullet, A_\circ, \bullet\eta^\circ)) \\
&= \bullet(A_\bullet \times_{\bullet A_\circ} A_\circ) && = \circ(A_\bullet \times_{\bullet A_\circ} A_\circ) \\
&= \bullet A_\bullet \times_{\bullet\bullet A_\circ} \bullet A_\circ && = \circ A_\bullet \times_{\circ\bullet A_\circ} \circ A_\circ && \text{(Lemma 2.14)} \\
&= \bullet A_\bullet \times_{\bullet A_\circ} \bullet A_\circ && = 1 \times_1 \circ A_\circ \\
&= \bullet A_\bullet && = \circ A_\circ \\
&= A_\bullet && = A_\circ
\end{aligned}$$

using the fact that $\circ\bullet A = 1$. □

Corollary 2.16 (✋). *Every $f : A \multimap B$ consists of a concrete homomorphism $f_\bullet : \bullet A \multimap \bullet B$ and an abstract homomorphism $f_\circ : \circ A \multimap \circ B$ that cohere (formally, $\bullet\eta^\circ_B \circ f_\bullet = \bullet(f_\circ \circ \eta^\circ_A)$).*

While the fracture and gluing result above depends centrally on modalities, much of the remainder of this work can be achieved without the direct use of modalities. Analogous to Definition 2.5, it is possible to treat any homomorphism as an abstraction homomorphism.

Definition 2.17 (✋). Define $\mathbf{Abstraction}(\alpha : A_\top \multimap A_{\mathrm{abs}}) \coloneqq \mathbf{Glue}(\bullet A_\top, \circ A_{\mathrm{abs}}, \bullet(\eta^\circ \circ \alpha))$.

Corollary 2.18 (✋). *By Theorem 2.15, it is the case that $A = \mathbf{Abstraction}(\mathrm{id}_A : A \multimap A)$ for all A.*

Lemma 2.19 (✋). *To define a map $f : \mathbf{Abstraction}(\alpha : A_\top \multimap A_{\mathrm{abs}}) \multimap \mathbf{Abstraction}(\beta : B_\top \multimap B_{\mathrm{abs}})$, it suffices by Corollary 2.16 to define a pair of maps $f_\top : A_\top \multimap B_\top$ and $f_{\mathrm{abs}} : A_{\mathrm{abs}} \multimap B_{\mathrm{abs}}$ such that*

$$\begin{array}{ccc}
A_\top & \xrightarrow{f_\top} & B_\top \\
{\scriptstyle\alpha}\downarrow & = & \downarrow{\scriptstyle\beta} \\
A_{\mathrm{abs}} & \xrightarrow[f_{\mathrm{abs}}]{} & B_{\mathrm{abs}}
\end{array}$$

using $f_\bullet = \bullet f_\top$ and $f_\circ = \circ f_{\mathrm{abs}}$.

The payoff of this construction, building on the insight of Grodin and Harper [2024], is that every type A contains an abstraction homomorphism indicating how much potential is stored within. Moreover, every homomorphism $A \multimap B$ contains proofs of both abstraction and the conservation of potential, *synthetically* reconstructing the physicist's view of amortized analysis.

Example 2.20 (✋). Adapting Example 2.8, we now build a cost-aware, ephemeral variant of the batched queue data structure that includes both abstraction and potential. The homomorphism that abstracts a batched queue (l_1, l_2) as a single list $\chi\ (l_1, l_2) \coloneqq l_1 +\!\!+ \mathit{reverse}\ l_2$ and emits its potential $\Phi\ (l_1, l_2) \coloneqq |l_2|$ can be constructed as follows:

$$\begin{aligned}
&\alpha : \mathbf{F}(\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}) \multimap \mathbf{F}(\textsc{List}\ \mathbb{N}) \\
&\alpha\ (\mathbf{ret}\ (l_1, l_2)) \coloneqq \mathbf{charge}\langle\Phi\ (l_1, l_2)\rangle(\mathbf{ret}\ (\chi\ (l_1, l_2)))
\end{aligned}$$

From this homomorphism α, we may build a type that contains all of this data, including the pair-of-lists implementation type, the single-list specification type, the value-level function χ, and the potential function Φ:

$$A \coloneqq \mathbf{Abstraction}(\alpha : \mathbf{F}(\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}) \multimap \mathbf{F}(\textsc{List}\ \mathbb{N})).$$

Using this type A, we may implement queue operations; each consisting of a concrete aspect (on the pair-of-lists type, annotated with realistic costs) and an abstract aspect (on the single-list type, annotated with amortized costs), linked by α in both behavior (up to χ) and cost (up to Φ).

To define $\mathit{enqueue} : \mathbb{N} \to A \multimap A$, it suffices by Lemma 2.19 to give the true code $\mathit{enqueue}_\top$ and an abstract mathematical model $\mathit{enqueue}_{\text{abs}}$ that cohere up to α, the proof of which includes both abstraction and amortization. Assuming a cost model that counts recursive calls, the true enqueue algorithm is defined as

$$\begin{aligned}
&\mathit{enqueue}_\top : \mathbb{N} \to \mathbf{F}(\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}) \multimap \mathbf{F}(\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}) \\
&\mathit{enqueue}_\top\ n\ (\mathbf{ret}\ (l_1, l_2)) \coloneqq \mathbf{ret}\ (l_1, n :: l_2),
\end{aligned}$$

without any cost annotations. On the other hand, the abstract specification is defined as

$$\begin{aligned}
&\mathit{enqueue}_{\text{abs}} : \mathbb{N} \to \mathbf{F}(\textsc{List}\ \mathbb{N}) \multimap \mathbf{F}(\textsc{List}\ \mathbb{N}) \\
&\mathit{enqueue}_{\text{abs}}\ n\ (\mathbf{ret}\ l) \coloneqq \mathbf{charge}\langle 1\rangle(\mathbf{ret}\ (l +\!\!+ [n])),
\end{aligned}$$

describing a client-facing *amortized cost interface* with one unit of cost to cohere with the potential function (and in anticipation of an impending linear-cost dequeue). The remainder of the enqueue implementation is the proof that the true code coheres with the abstract amortized specification:

$$\begin{array}{ccc}
\mathbf{F}(\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}) & \xrightarrow{\mathit{enqueue}_\top\ n} & \mathbf{F}(\textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}) \\
{\scriptstyle\alpha}\downarrow & = & \downarrow{\scriptstyle\alpha} \\
\mathbf{F}(\textsc{List}\ \mathbb{N}) & \xrightarrow[\mathit{enqueue}_{\text{abs}}\ n]{} & \mathbf{F}(\textsc{List}\ \mathbb{N})
\end{array}$$

Equationally, the proof obligation expressed by this square holds by the following reasoning:

$$\begin{aligned}
\alpha\ (\mathit{enqueue}_\top\ n\ (\mathbf{ret}\ (l_1, l_2))) &= \alpha\ (\mathbf{ret}\ (l_1, n :: l_2)) \\
&= \mathbf{charge}\langle |n :: l_2|\rangle(\mathbf{ret}\ (l_1 +\!\!+ \mathit{reverse}\ (n :: l_2))) \\
&= \mathbf{charge}\langle |l_2| + 1\rangle(\mathbf{ret}\ (l_1 +\!\!+ \mathit{reverse}\ l_2 +\!\!+ [n])) \\
&= \mathbf{charge}\langle |l_2|\rangle(\mathbf{charge}\langle 1\rangle(\mathbf{ret}\ (l_1 +\!\!+ \mathit{reverse}\ l_2 +\!\!+ [n]))) \\
&= \mathbf{charge}\langle |l_2|\rangle(\mathit{enqueue}_{\text{abs}}\ n\ (\mathbf{ret}\ (l_1 +\!\!+ \mathit{reverse}\ l_2))) \\
&= \mathit{enqueue}_{\text{abs}}\ n\ (\mathbf{charge}\langle |l_2|\rangle(\mathbf{ret}\ (l_1 +\!\!+ \mathit{reverse}\ l_2))) \qquad (\star) \\
&= \mathit{enqueue}_{\text{abs}}\ n\ (\alpha\ (\mathbf{ret}\ (l_1, l_2)))
\end{aligned}$$

The indicated step ($\star$) holds because $\mathit{enqueue}_{\text{abs}}$ is a homomorphism of cost algebras. Notice that in addition to showing that χ is preserved for abstraction [Grodin et al. 2026], this derivation verifies that potential is conserved [Grodin and Harper 2024]: specifically, it includes the fact that

$$0 + \Phi(l_1, n :: l_2) = \Phi(l_1, l_2) + 1,$$

where 0 is the true cost and 1 is the amortized cost. The $\mathit{empty} : \mathbf{U}A$ and $\mathit{dequeue} : A \multimap \mathbb{N} \rtimes A$ operations can be similarly defined, following Grodin et al. [2026]. ⌟

2.3 Potential and Abstraction, Independently

Although the abstraction homomorphisms built into types generally include both abstraction and potential, these concerns need not be considered simultaneously as above. As an additional convenience, it is possible to build a potential function $\Phi : X \to \mathbb{C}$ into the type $\mathbf{F}X$.

Definition 2.21 (✋). To render a potential function $\Phi : X \to \mathbb{C}$ as a type, define

$$\mathbf{Potential}(\Phi : X \to \mathbb{C}) \coloneqq \mathbf{Abstraction}(\varphi(\Phi) : \mathbf{F}X \multimap \mathbf{F}X)$$

where $\varphi(\Phi) \coloneqq \lambda\ (\mathbf{ret}\ x) \multimap \mathbf{charge}\langle\Phi(x)\rangle(\mathbf{ret}\ x)$.

Lemma 2.22 (✋). *Let $A \coloneqq \mathbf{Potential}(\Phi_X : X \to \mathbb{C})$ and $B \coloneqq \mathbf{Potential}(\Phi_Y : Y \to \mathbb{C})$. To define a homomorphism of type $A \multimap B$, it suffices to provide a function $f : X \to Y$, a true cost function $c_\top : X \to \mathbb{C}$, and an amortized cost function $c_{\mathrm{abs}} : X \to \mathbb{C}$ such that potential is conserved:*

$$c_\top(x) + \Phi_Y(f(x)) = \Phi_X(x) + c_{\mathrm{abs}}(x).$$

Proof. By Lemma 2.19, with $f_i\ (\mathbf{ret}\ x) \coloneqq \mathbf{charge}\langle c_i(x)\rangle(\mathbf{ret}\ (f(x)))$. □

Corollary 2.23 (✋). *Let $B \coloneqq \mathbf{Abstraction}(\beta : B_\top \multimap B_{\mathrm{abs}})$ and let $A : \mathcal{C}$ be arbitrary.*

(1) *To define a homomorphism of type $f : A \multimap B$, it suffices to provide $f_\top : A \multimap B_\top$.*
(2) *To define a homomorphism of type $g : B \multimap A$, it suffices to provide $g_{\mathrm{abs}} : B_{\mathrm{abs}} \multimap A$.*

Proof. By Corollary 2.18, it is the case that $A = \mathbf{Abstraction}(\mathrm{id}_A : A \multimap A)$. Then, the results follow by Lemma 2.19, with $f_{\mathrm{abs}} \coloneqq \beta \circ f_\top$ and $g_\top \coloneqq g_{\mathrm{abs}} \circ \beta$. □

Using this construction, the amortized analysis and abstraction of Example 2.20 may be achieved sequentially rather than simultaneously.

Example 2.24. First, define a type to incorporate only the potential function for batched queues:

$$B \coloneqq \mathbf{Potential}(\lambda\ (l_1, l_2) \to |l_2| : \textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N} \to \mathbb{C}).$$

To define $enqueue_\top : \mathbb{N} \to B \multimap B$, it suffices by Lemma 2.22 to define

$$f\ n\ (l_1, l_2) \coloneqq (l_1, n :: l_2) \qquad c_\top\ n\ (l_1, l_2) \coloneqq 0 \qquad c_{\mathrm{abs}}\ n\ (l_1, l_2) \coloneqq 1$$

and prove that the conservation of potential equation holds. This definition exports an amortized cost specification saying that the enqueue operation takes 1 amortized cost; however, the behavioral aspect of the specification still reveals that the data structure is implemented as a pair of lists. To remedy this, we define a type whose concrete part is inherited from B itself, but whose abstract part exports the list-based representation (and maintains the amortized cost specification). Let

$$A \coloneqq \mathbf{Abstraction}(\alpha : B \multimap \mathbf{F}(\textsc{List}\ \mathbb{N})),$$

where α can be defined using Corollary 2.23 given only the lifting of the abstraction function

$$\lambda\ (l_1, l_2) \to l_1 +\!\!+ reverse\ l_2 : \textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N} \to \textsc{List}\ \mathbb{N}.$$

To define $enqueue : \mathbb{N} \to A \multimap A$, it suffices to combine $enqueue_\top$ from above with $enqueue_{\mathrm{abs}}$ from Example 2.20. Thus, we were able to first perform the amortized analysis of batched queues and subsequently overlay a coherent abstract mathematical data specification. ⌟

3 LOSS OF ENERGY DUE TO ABSTRACTION

As developed by Grodin et al. [2026, §3], the abstract phase facilitates modularity, guaranteeing that client verifications do not depend on library implementation details. In this section, we apply such reasoning to the present setting to facilitate modular verification of amortized cost. Then, building on the approach of Grodin et al. [2026, §4.5], we show how to accommodate inequality in the conservation of potential condition, which improves modularity in the cost-aware setting.

3.1 Modularity and Amortized Cost Interfaces

The abstract phase makes it possible to uniformly isolate the public-facing specification associated with any type or program, allowing correctness theorems to be stated under the explicit assumption that concrete implementation details have been ignored. This facilitates modular verification: if clients only prove theorems under the assumption of **abs**, there is no trouble modifying a library so long as it continues to maintain the same public-facing specification. Phase-sensitive restrictions can be made using the notion of a *specification type*.

Definition 3.1 (Specification Type, Grodin et al. 2026). Let $¶ : \mathcal{V}$ be a proposition. The *$¶$-phase specification type* for a specification $x_\circ : ¶ \to X$ is the type

$$\{X \mid ¶ \hookrightarrow x_\circ\} \coloneqq \sum_{x:X} ((\_ : ¶) \to x = x_\circ(\_))$$

describing all the inhabitants $x : X$ that cohere with $x_\circ$ in the phase $¶$.[6]

Using a specification type, we will define an interface of ephemeral queues. Although implementations may perform the cost effect, the interface for ephemeral queues should only restrict abstract *behavior*. This can be achieved using the *behavioral phase* proposition **beh** of Calf [Niu et al. 2022], which isolates the behavior of a program from its cost by erasing the cost effect:

$$\mathbf{beh} \to \mathbf{charge}\langle c\rangle(e) = e.$$

When combined with abstraction, **beh** is axiomatized to imply **abs** [Grodin et al. 2026].

Example 3.2 (Ephemeral Queue Interface). Say an ephemeral pre-queue is a computation type $A : C$ equipped with standard queue operations:

$$\textsc{PreQueue}_C \coloneqq \sum_{A:C} (\mathsf{empty} : \mathbf{U}A) \times (\mathsf{enqueue} : \mathbb{N} \to A \multimap A) \times (\mathsf{dequeue} : A \multimap \mathbb{N} \rtimes A).$$

The interface of ephemeral queues, $\textsc{Queue}_C$, refines $\textsc{PreQueue}_C$ with a *behavioral* specification:

$$\{\textsc{PreQueue}_C \mid \mathbf{beh} \hookrightarrow (\mathbf{F}(\textsc{List}\ \mathbb{N}), \mathbf{ret}\ [], (\lambda\ n \to \lambda\ (\mathbf{ret}\ l) \multimap \mathbf{ret}\ (l +\!\!+ [n])), \mathit{uncons})\}.$$

This restriction completely determines the abstract aspect of an implementation, aside from cost. The batched queues of Example 2.20 are a valid implementation of this interface, by construction: erasing the costs of the interface-level components recovers precisely the given specification. ⌟

Although this behavioral restriction entirely pins down the behavior of an implementation, it makes no claims about the cost of an implementation. In order to reveal information about the cost, it suffices to *refine* the interface $\textsc{Queue}_C$ with stronger guarantees, such as an **abs**-phase restriction revealing an *amortized cost specification*.

Example 3.3 (Ephemeral Queue Cost Interface 🤌). Building on the ephemeral queue interface of Example 3.2, consider the following refinement, noting the use of **abs** instead of **beh**:

$$\{\textsc{PreQueue} \mid \mathbf{abs} \hookrightarrow (\mathbf{F}(\textsc{List}\ \mathbb{N}), \mathbf{ret}\ [], (\lambda\ n \to \lambda\ (\mathbf{ret}\ l) \multimap \mathbf{charge}\langle 1\rangle(\mathbf{ret}\ (l +\!\!+ [n]))), \mathit{uncons})\}.$$

Beyond the behavioral guarantees consistent with $\textsc{Queue}_C$, this interface exports amortized costs of the operations, classifying queues whose enqueue operation has amortized cost of 1 and whose empty and dequeue operations (implicitly, by lack of cost annotation) have amortized cost of 0. By construction, the batched queue implementation of Example 2.20 inhabits this refinement. ⌟

[6] Per Grodin et al. [2026], the notation is inspired by extension types [Riehl and Shulman 2017], but the equality is not definitional.

3.2 Amortized Upper Bounds as Lax Commutative Squares

The batched queue data structure has the property that its amortized cost model exactly matches the implementation. In particular, its analysis satisfies a strict conservation principle:

$$c_{\top}(x) + \Phi(f(x)) = \Phi(x) + c_{\mathrm{abs}}(x).$$

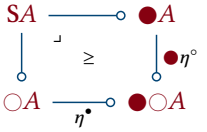
However, such exactness is rare: the true cost of an algorithm often depends on private implementation details, and the amortized cost is thus merely an upper bound of the true cost. To this end, it is necessary to relax the strict equality-based commutativity of the squares built into homomorphisms to merely a lax inequality. This representing the idea of energy not being perfectly conserved:

$$c_{\top}(x) + \Phi(f(x)) \leq \Phi(x) + c_{\mathrm{abs}}(x).$$

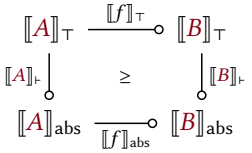
In physics, energy can be lost due to sources such as friction; in this setting, amortized cost bounds can be weakened due to the demands of *abstraction*.

In order to support weakening of cost bounds via lax commutative squares, we follow Grodin et al. [2026] and use the *sealing monad* $\mathbf{S}$, defined here on computation types as a comma object:

$$\mathbf{S} : \mathcal{C} \to \mathcal{C}$$
$$\mathbf{S}A = \bullet A \times^{\leq}_{\bullet \bigcirc A} \bigcirc A$$



Semantically[7], a Kleisli map $f : A \multimap \mathbf{S}B$ is exactly a lax commutative square:



Thus, we define a type of lax homomorphisms[8] that allows weakening of cost and potential:

$$A \nrightarrow B \coloneqq A \multimap \mathbf{S}B.$$

Remark 3.4. Grodin et al. [2026] added a sealing effect to computation types (at the same level as the cost effect) and used the sealing monad as a semantics. However, this interferes with our proof of fracture and gluing (Theorem 2.15); hence, we make the sealing monad user-specified.

Lemma 3.5. *In the abstract phase (*i.e.*, assuming* **abs***), it is the case that* $\mathbf{S}A = A$.

Example 3.6. The splay tree data structure [Sleator and Tarjan 1985b] is a purely functional implementation of arrays with an amortized logarithmic-time lookup operation. The precise cost depends on internal implementation details and is not exactly representable in the abstract phase; to accommodate this laxity, one may use a lax homomorphism, as verified analytically in Calf by Kebuladze [2025]. In the present synthetic setting, the implementation type of splay trees can be

$$A \coloneqq \mathbf{S}(\mathbf{Abstraction}(\alpha : \mathbf{F}(\textsc{Tree}\ \mathbb{N}) \multimap \mathbf{F}(\textsc{List}\ \mathbb{N}))),$$

where α is the abstraction homomorphism described by Kebuladze. By Lemma 3.5, this implementation type is revealed as $\mathbf{F}(\textsc{List}\ \mathbb{N})$ in the abstract phase, facilitating modularity. ⌟

[7] In the presheaf semantics, the synthetic sealing monad is interpreted as the free opfibration 2-monad [Street 1974].
[8] Note that the laxity here is of the square induced by the abstract phase, *not* of the cost algebra homomorphism.

4 THE BANKER'S VIEW

In contrast to the physicist's potential, the banker imagines *credits* and *debits* being stored within data structures. In the original work on amortized analysis, Tarjan [1985] notes that credits and debits will not appear in programs themselves. However, in the present setting of cost verification, credits and debits will indeed appear both in types and terms. To render the banker's view within type theory, we will define notions of credit and debit internally in terms of abstraction.

4.1 Credits and Debits

We now implement credit and debit within the type theory using abstraction homomorphisms.

4.1.1 The Credit Operator. Because potential is analogous to credit, we may store c credits on a data structure by emitting c additional units of cost in the included abstraction homomorphism.

Definition 4.1 (✋). The *credit operator* annotates a type A with c additional credits by constructing a variant of A with an abstraction homomorphism that emits c additional units of cost:

$$\triangleright : \mathbb{C} \to C \to C$$
$$\triangleright^c A := \mathbf{Abstraction}(\mathbf{charge}_A\langle c\rangle : A \multimap A)$$

Remark 4.2. By Definition 2.11, we have $\mathbf{charge}_A\langle c\rangle : \mathbf{U}A \to \mathbf{U}A$. The ability to lift the cost effect to a homomorphism $A \multimap A$ is precisely commutativity of the cost monoid $(\mathbb{C}, 0, +)$.

Remark 4.3. The credit operator has the effect not of replacing the existing abstraction homomorphism contained within A, but rather increasing its cost by c. In the Kripke semantics:

$$\triangleright^c \begin{pmatrix} [\![A]\!]_\top \\ \downarrow \alpha \\ [\![A]\!]_{\mathrm{abs}} \end{pmatrix} := \begin{pmatrix} [\![A]\!]_\top \\ \downarrow \mathbf{charge}_{A_\top}\langle c\rangle \circ \alpha \\ [\![A]\!]_{\mathrm{abs}} \end{pmatrix}$$

Lemma 4.4 (✋). *Credits are* ghost data, *invisible in both the concrete program and the abstract specification, only appearing to mediate between the two:* $\bullet(\triangleright^c A) = \bullet A$ *and* $\circ(\triangleright^c A) = \circ A$.

Lemma 4.5. *For $c' \geq c$, the credit operator admits a weakening map $\triangleright^{c'} A \multimap \triangleright^c A$ that wastes credits.*

Lemma 4.6 (✋). *The credit operator satisfies $\triangleright^0 A = A$ and $\triangleright^{c_1+c_2} A = \triangleright^{c_1}\triangleright^{c_2} A$.*

Definition 4.7 (✋). Using Corollary 2.23 with $f_\top, g_{\mathrm{abs}} : A \multimap A$ being the identity function, define

$$\mathsf{save}\langle c\rangle : A \multimap \triangleright^c A \qquad\qquad \mathsf{spend}\langle c\rangle : \triangleright^c A \multimap A.$$

The $\mathsf{save}\langle c\rangle$ operation incurs no true cost, but it instead reports c units of abstract cost to clients in the abstract phase. Dually, the $\mathsf{spend}\langle c\rangle$ operation truly incurs c cost, but it reports no cost to clients in the abstract phase. This can be visualized in the Kripke semantics, where the $\mathsf{save}\langle c\rangle$ and $\mathsf{spend}\langle c\rangle$ operations are interpreted as the following vertical composites of squares, respectively:

$$\begin{array}{ccc} [\![A]\!]_\top & = & [\![A]\!]_\top \\ {\scriptstyle\alpha}\downarrow & = & \downarrow{\scriptstyle\alpha} \\ [\![A]\!]_{\mathrm{abs}} & = & [\![A]\!]_{\mathrm{abs}} \\ \| & = & \downarrow{\scriptstyle\mathbf{charge}\langle c\rangle} \\ [\![A]\!]_{\mathrm{abs}} & \xrightarrow[\mathbf{charge}\langle c\rangle]{} & [\![A]\!]_{\mathrm{abs}} \end{array} \qquad\qquad \begin{array}{ccc} [\![A]\!]_\top & \xrightarrow{\mathbf{charge}\langle c\rangle} & [\![A]\!]_\top \\ {\scriptstyle\alpha}\downarrow & = & \downarrow{\scriptstyle\alpha} \\ [\![A]\!]_{\mathrm{abs}} & \xrightarrow{\mathbf{charge}\langle c\rangle} & [\![A]\!]_{\mathrm{abs}} \\ {\scriptstyle\mathbf{charge}\langle c\rangle}\downarrow & = & \| \\ [\![A]\!]_{\mathrm{abs}} & = & [\![A]\!]_{\mathrm{abs}} \end{array}$$

Reading the diagrams from left to the right, saving credits moves cost from the abstract specification into the output type, and spending credits moves cost from the input type to the true code.

Lemma 4.8 (✋). *For all c,* $\mathsf{spend}\langle c\rangle \circ \mathsf{save}\langle c\rangle = \mathbf{charge}_A\langle c\rangle$.

Remark 4.9. In the language of double category theory, the bottom halves of the above composites with Lemma 4.8 render cost and potential/credits as *companions* [Grandis and Paré 2004], representing the idea that cost and potential/credits are the same idea but in different "dimensions".

Lemma 4.10. *Recalling Definition 2.21, the following types are equivalent:*

$$\mathbf{Potential}(\Phi : X \to \mathbb{C}) = (x : X) \rtimes \triangleright^{\Phi(x)} \top.$$

This formally connects the physicist's view and the banker's view: the type containing the potential function Φ is equivalent to the type representing a value $x : X$ stored alongside $\Phi(x) : \mathbb{C}$ credits.

4.1.2 The Debit Operator. Using the credit operator, we may define its dual, the debit operator, which as an input indicates that some credits are owed.

Definition 4.11 (✋). The *debit operator* annotates a type A with an opportunity to make use of c credits, achieved via a lax homomorphism assuming credits:

$$\triangleleft : \mathbb{C} \to C \to C$$
$$\triangleleft^c A \coloneqq \triangleright^c \top \multimap A$$

We choose to use a lax homomorphism here in order to make debits provide an opportunity rather than a burden; with a strict homomorphism $\triangleright^c \top \multimap A$, all assumed credits must be spent.

Lemma 4.12 (✋). *For all $c : \mathbb{C}$, the debit and credit operators are adjoint: $\triangleright^c \dashv \triangleleft^c$. The unit $A \multimap \triangleleft^c \triangleright^c A$ takes out a loan, and the counit $\triangleright^c \triangleleft^c A \multimap A$ pays off a loan.*

Using debits, it is possible to implement advanced amortized data structures, such as the implicit queues of Okasaki [1999, §11] as shown by Danielsson [2008] and Rajani [2020].

4.2 Credit-Carrying Lists

Using the credit operator, we may build inductive data structures that contain credits, as is standard in the banker's method. First, we develop two variants of lists: one that stores credits linear in the length of the list, and one that stores credits quadratic in the length of the list.

Definition 4.13 (✋). Let $\textsc{CList}_1^c\ A \coloneqq \textsc{List}\ (\triangleright^c A)$ classify lists of elements of type A in which each element is accompanied by c credits.

Corollary 4.14. *In the abstract phase,* $\textsc{CList}_1^c\ (\mathbf{F}X) = \textsc{List}\ (\mathbf{F}X) = \mathbf{F}(\textsc{List}\ X)$.

Example 4.15. Using the induction principle for the type of lists with linear credits, we may define a linear-time list reverse algorithm that uses the stored credits as follows.

$$\begin{array}{ll}
\mathit{revAppend} : \textsc{CList}_1^1\ (\mathbf{F}X) \multimap \textsc{List}\ X \rightharpoonup \mathbf{F}(\textsc{List}\ X) & \mathit{reverse} : \textsc{CList}_1^1\ (\mathbf{F}X) \multimap \mathbf{F}(\textsc{List}\ X) \\
\mathit{revAppend}\ [\,] \coloneqq \lambda\ \mathit{acc} \rightharpoonup \mathbf{ret}\ \mathit{acc} & \mathit{reverse}\ l \coloneqq \mathit{revAppend}\ l\ [\,] \\
\mathit{revAppend}\ (a :: l) \coloneqq \lambda\ \mathit{acc} \rightharpoonup & \\
\quad \mathbf{let}\ (\mathbf{ret}\ x) \coloneqq \mathsf{spend}\langle 1\rangle(a)\ \mathbf{in} & \\
\quad \mathit{revAppend}\ l\ (x :: \mathit{acc}) &
\end{array}$$

Note that the only cost effect occurs within $\mathsf{spend}\langle 1\rangle$; this means that abstractly, both *revAppend* and *reverse* are zero-cost, because the credits required are pre-paid into the input list. ⌟

Using this credit-assuming reverse function, we may implement batched queues using the banker's view, as the potential function assigned one unit of potential per element of the back list.

Example 4.16. Let $B \coloneqq \textsc{List}\ \mathbb{N} \rtimes \textsc{CList}_1^1\ (\mathbf{F}\mathbb{N})$ describe pairs of lists of natural numbers where each element of the second list is equipped with one credit. To define $\mathit{enqueue} : \mathbb{N} \to B \multimap B$, we first define an auxiliary function to accept a credit, to be stored alongside the new list element:

$$\mathit{enqueue}' : \triangleright^1(\mathbb{N} \rtimes B) \multimap B$$

Then, $\mathit{enqueue} \coloneqq \lambda\, n \to \lambda\, b \multimap \mathit{enqueue}'\ (\mathrm{save}\langle 1\rangle(n, b))$, using the $\mathrm{save}\langle 1\rangle$ derived form to pre-pay for the credit, revealed as amortized cost in the abstract phase. The other operations are similar, where *dequeue* makes use of *reverse* from Example 4.15. ⌟

The type B does not meet the queue interface of Example 3.2, as it does not perform abstraction to masquerade as a single list. Following Example 2.24, it is possible to induce this abstraction using a function $\alpha : B \multimap \mathbf{F}(\textsc{List}\ \mathbb{N})$. Alternatively, we may use a *phased quotient* [Grodin et al. 2026, §2.3] to abstract implicitly while maintaining the locality of the banker's method.

Example 4.17. The representation type B of Example 4.16 may be augmented to be suitably abstract as an implementation of queues by applying a quotient in the abstract phase:

data $A : C$ **where**
  $\mathrm{inj} : B \multimap A$
  $\mathrm{tilt} : \mathbf{abs} \to (x : \mathbb{N}) \to (l_1\ l_2 : \textsc{List}\ \mathbb{N}) \to \mathrm{inj}\ (l_1 +\!\!+ [x], \mathbf{ret}\ l_2) = \mathrm{inj}\ (l_1, \mathbf{ret}\ (l_2 +\!\!+ [x]))$

Implicitly, we use Corollary 4.14. This type A is thus equivalent in the abstract phase to $\mathbf{F}(\textsc{List}\ \mathbb{N})$. When the operations sketched in Example 4.16 are shown to preserve this abstract quotient, this structure implements the ephemeral queue interfaces of Examples 3.2 and 3.3. ⌟

Generalizing linear-credit lists of the previous section, we demonstrate the case of lists carrying linear and triangular (*i.e.*, $c_1 n + c_2 \binom{n}{2}$) credits.

Definition 4.18 (🤲). Define $\textsc{CList}_2^{(c_1,c_2)}\ A$ to be the following inductive type family:

data $\textsc{CList}_2^{(c_1,c_2)}\ A : C$ **where**
  $[] : \top \multimap \textsc{CList}_2^{(c_1,c_2)}\ A$
  $\_ :: \_ : \triangleright^{c_1} A \otimes \textsc{CList}_2^{(c_2+c_1,c_2)}\ A \multimap \textsc{CList}_2^{(c_1,c_2)}\ A$

Note that this inductive family varies c_1 and is thus *not* representable via $\textsc{List}\ (-)$.

Corollary 4.19. *In the abstract phase,* $\textsc{CList}_2^{(c_1,c_2)}\ (\mathbf{F}X) = \textsc{List}\ (\mathbf{F}X) = \mathbf{F}(\textsc{List}\ X)$.

Using this variety of credit-carrying list, it is possible to implement pre-paid versions of quadratic-cost algorithms, such as insertion sort. We return to this line of development in Section 5.

4.3 Credit-Carrying Trees

Example 4.20. Tarjan [1985] established an amortized analysis of red-black trees [Guibas and Sedgewick 1978] in which each black node contains credits computed based on the color of its child nodes. We may represent this construction in the banker's view as follows, where $0 \le c(x_1, x_2) \le 2$ is the number of credits to be stored at a black node with child nodes of colors x_1 and x_2.

data $\textsc{RBTree}\ (A : C)\ (x : \textsc{Color})\ (h : \mathbb{N}) : C$ **where**
  $\mathrm{empty} : \top \multimap \textsc{RBTree}\ A\ \mathrm{black}\ 0$
  $\mathrm{red} : \textsc{RBTree}\ A\ \mathrm{black}\ h \otimes A \otimes \textsc{RBTree}\ A\ \mathrm{black}\ h \multimap \textsc{RBTree}\ A\ \mathrm{red}\ h$
  $\mathrm{black} : \triangleright^{c(x_1,x_2)}(\textsc{RBTree}\ A\ x_1\ h \otimes A \otimes \textsc{RBTree}\ A\ x_2\ h) \multimap \textsc{RBTree}\ A\ \mathrm{black}\ (1 + h)$

This construction can be adapted to support the appropriate abstraction using the techniques used on batched queues, following Grodin et al. [2026]. ⌟

Example 4.21. The amortized splay tree [Sleator and Tarjan 1985b] data structure stores $\lceil \lg n \rceil$ credits at each node whose subtree is of size n. Such credits can be represented in the following inductive type, where $c(n_1, n_2) \coloneqq \lceil \lg(n_1 + 1 + n_2) \rceil$.

data SplayTree $(A : C)$ $(n : \mathbb{N}) : C$ **where**
  leaf : $\top \multimap$ SplayTree A 0
  node : $\rhd^{c(n_1,n_2)}$(SplayTree A $n_1 \otimes A \otimes$ SplayTree A n_2) $\multimap$ SplayTree A $(n_1 + 1 + n_2)$

The credits associated to splay trees are notoriously difficult to annotate (and analyze automatically) in AARA [Hofmann et al. 2022]. However, in the dependent setting of the present work, it is straightforward to include credits just as originally described by Sleator and Tarjan. ⌟

5 GIRALF: A GRADED, INFERENTIAL, RESOURCE-AWARE LOGICAL FRAMEWORK

In order to streamline the development of programs involving credits and debits, we now define a graded substructural type theory called *Giralf* overlaid upon the existing dependent type theory of Calf; drawing inspiration from AARA [Hoffmann and Jost 2022; Hofmann and Jost 2003], the credit and debit type operators internalize the graded judgmental structure of Giralf. The types of Giralf are taken from Calf, and thus Giralf programs semantically constitute a well-behaved subclass of Calf programs, demonstrating that AARA-like programs exist as a sub-language of the computations of Calf. Furthermore, by adapting the techniques of AARA, programs written in the Giralf language are amenable to a form of automated cost inference; thus, as a semantic sub-language of Calf, Giralf facilitates the integration of manual and automatic cost verification.

5.1 A Graded Syntax (🖐)

The syntax of Giralf is similar to that of Das et al. [2021], but with two major differences:

(1) *Dependency.* Giralf admits dependency of substructural resources on structural values, like in the dependent linear/non-linear type theory of Krishnaswami et al. [2015]. For example, credit-carrying types refer to structural values of type $\mathbb{C}$. This is important for handling indexed inductive types, such as the quadratic-credit lists of Section 4.2.
(2) *Recursion.* It is typical for AARA-like languages to admit unbounded recursion. In Giralf, we only consider the structural recursion principles induced by inductive types.[9]

We now define Giralf, whose types are inherited from Calf (including those of Section 4). The typing judgment $\Gamma \mid \Delta \vdash^q e : A$ means that given a structural context Γ, a linear context $\Delta = A_1, \cdots, A_n$ dependent on Γ, and additional credits $q : \mathbb{C}$ also dependent on Γ, the program e has type A. For readability, we leave Γ implicit, only notating the extension beyond the ambient Γ.

We first define the rules that apply uniformly over types. The variable (identity) and let-binding (cut) rules are standard from graded type theory:

$$\frac{}{a : A \vdash^q a : A} \qquad \frac{q \geq q_1 + q_2 \qquad \Delta_1 \vdash^{q_1} e_1 : A \qquad \Delta_2, a : A \vdash^{q_2} e_2 : B}{\Delta_1, \Delta_2 \vdash^q \mathbf{let}\, a = e_1 \,\mathbf{in}\, e_2 : B}$$

The rule for spending credits follows AARA [Das et al. 2021; Hoffmann and Jost 2022; Hofmann and Jost 2003]. Justified by having sufficient credits in the context, this construct incurs cost, to be annotated on programs as a cost model.

$$\frac{q \geq p + q' \qquad \Delta \vdash^{q'} e : A}{\Delta \vdash^q \mathbf{spend}\langle p \rangle(e) : A}$$

[9] It is possible to treat unbounded recursion in Calf as an effect [Niu and Harper 2022]; however, it is not clear that this effect is compatible with the fracture and gluing principle central to this work.

5.1.1 Standard Linear Types. The rules for standard linear types are as usual. Of note, Giralf includes negative types, such as lazy products (like λ-amor [Rajani et al. 2021] but unlike AARA):

$$\frac{\Delta \vdash^q e_1 : A_1 \qquad \Delta \vdash^q e_2 : A_2}{\Delta \vdash^q (e_1, e_2) : A_1 \times A_2} \qquad \frac{\Delta \vdash^q e : A_1 \times A_2}{\Delta \vdash^q \mathbf{proj}_i\ e : A_i}$$

We now turn our attention to types that interact with the credit context.

5.1.2 Credit and Debit. The rules for the credit operator are analogous to those given by Das et al. [2021] and Rajani et al. [2021], internalizing credits from the credit context as a type former.

$$\frac{q \geq p + q' \qquad \Delta \vdash^{q'} e : A}{\Delta \vdash^q \mathbf{store}\langle p \rangle(e) : \triangleright^p A} \qquad \frac{q \geq q_1 + q_2 \qquad \Delta_1 \vdash^{q_1} e_1 : \triangleright^p A \qquad \Delta_2, a : A \vdash^{p+q_2} e_2 : B}{\Delta_1, \Delta_2 \vdash^q \mathbf{let\ store}(a) = e_1\ \mathbf{in}\ e_2 : B}$$

The rules for the debit operator are analogous to those given by Das et al. [2021].

$$\frac{q' \geq p + q \qquad \Delta \vdash^{q'} e : A}{\Delta \vdash^q \mathbf{get}\langle p \rangle(e) : \triangleleft^p A} \qquad \frac{q \geq p + q' \qquad \Delta \vdash^{q'} e : \triangleleft^p A}{\Delta \vdash^q \mathbf{pay}(e) : A}$$

5.1.3 Linear-Credit Lists. The AARA-like lists with linear credit described in Section 4.2 can be smoothly incorporated into Giralf, providing an elimination form in terms of structural recursion:

$$\frac{}{\cdot \vdash^q [] : \textsc{List}_1^p\ A} \qquad \frac{q \geq p + q_1 + q_2 \qquad \Delta_1 \vdash^{q_1} a : A \qquad \Delta_2 \vdash^{q_2} l : \textsc{List}_1^p\ A}{\Delta_1, \Delta_2 \vdash^q a :: l : \textsc{List}_1^p\ A}$$

$$\frac{\cdot \vdash^0 e_0 : B \qquad a : A, b : B \vdash^p e_1 : B \qquad \Delta \vdash^q e : \textsc{List}_1^p\ A}{\Delta \vdash^q \mathbf{foldr}[e_0; a.b.e_1](e) : B}$$

Using this recursion principle, it is possible to implement a variety of linear-time algorithms whose cost is pre-paid for by the available credits.

Lemma 5.1. *The following credit-aware* paramorphism *[Meertens 1992] is derivable:*

$$\frac{q \geq q_1 + q_2 \qquad \cdot \vdash^{q_2} e_0 : B \qquad a : A, b : \triangleleft^{q_2} B \times \textsc{List}_1^p\ A \vdash^{p+q_2} e_1 : B \qquad \Delta \vdash^{q_1} e : \textsc{List}_1^p\ A}{\Delta \vdash^q \mathbf{para}[e_0; a.b.e_1](e) : B}$$

This construction implicitly threads through credits available at the top level using the debit operator; moreover, using lazy products, it offers the choice between the recursive result and the current sublist.

Example 5.2 (☞). The main subroutine of insertion sort is definable in Giralf as a term

$$p : \mathbb{C}, x : \mathbb{N} \mid l : \textsc{List}_1^{1+p}\ (\mathbf{F}\mathbb{N}) \vdash^p \mathit{insert}\ x\ l : \textsc{List}_1^p\ (\mathbf{F}\mathbb{N})$$

using a paramorphism $\mathit{insert}\ x\ l \coloneqq \mathbf{para}[\mathbf{ret}\ x :: []; (\mathbf{ret}\ y).b.e_1]$ where

$$e_1 \coloneqq \mathbf{spend}\langle 1 \rangle(\mathbf{if}\ x \leq y\ \mathbf{then}\ (\mathbf{ret}\ x :: \mathbf{ret}\ y :: \mathbf{proj}_2\ b)\ \mathbf{else}\ (\mathbf{ret}\ y :: \mathbf{pay}(\mathbf{proj}_1\ b))).$$

In the base case, a singleton list containing only x is created. In the inductive case, if the new element x is smaller than the head element y, then x and y are placed on the front of the original list, recovered via $\mathbf{proj}_2\ b$; if x is larger than y, then y is placed on the front of the recursive call, passing down the credits that will eventually be attached to the new list node. ⌟

5.1.4 Quadratic-Credit Lists. The introduction rules for quadratic-credit lists are standard, following Hoffmann and Hofmann [2010a,b]:

$$\frac{}{\cdot \vdash^q [] : \textsc{CList}_2^{(p_1,p_2)}\ A} \qquad \frac{q \geq p_1 + q_1 + q_2 \qquad \Delta_1 \vdash^{q_1} a : A \qquad \Delta_2 \vdash^{q_2} l : \textsc{CList}_2^{(p_2+p_1,p_2)}\ A}{\Delta \vdash^q a :: l : \textsc{CList}_2^{(p_1,p_2)}\ A}$$

The elimination form, however, diverges from existing presentations due to the requirement here to use only bounded recursion. In particular, due to the status of quadratic-credit lists as an inductive family, their recursion principle makes nontrivial use of the structural context and dependency:

$$\frac{r : \mathbb{C} \mid \cdot \vdash^0 e_0 : B(r) \qquad r : \mathbb{C} \mid a : A, b : B(p_2 + r) \vdash^r e_1 : B(r) \qquad \Delta \vdash^q e : \textsc{CList}_2^{(p_1,p_2)}\ A}{\Delta \vdash^q \mathbf{foldr}\{r.B(r)\}[r.e_0; r.a.b.e_1](e) : B(p_1)}$$

This rule eliminates into a *family* of types $B : \mathbb{C} \to \mathcal{C}$ indexed by the linear credit coefficient; the family of outputs is required precisely because $\textsc{CList}_2^{(-,p_2)}\ A : \mathbb{C} \to \mathcal{C}$ is not an inductively-defined type, but rather an inductively-defined *family*. At the top level, the result type is $B(p_1)$, where p_1 is the linear coefficient of the list being eliminated. Inductively, both the base case and the inductive case must construct an element of type $B(r)$, where $r : \mathbb{C}$ is a freshly bound variable, required because the amount of linear credit changes inductively. This rule is in place of an ad-hoc rule for resource-polymorphic recursion [Hoffmann and Hofmann 2010a].

Example 5.3 (✋). The insertion sort algorithm, which in the worst case costs $\binom{n}{2}$ on a list of length n when counting comparisons, can be implemented as a term

$$l : \textsc{CList}_2^{(0,1)}\ (\mathbf{F}\mathbb{N}) \vdash^0 \mathit{isort}\ l : \textsc{CList}_1^0\ (\mathbf{F}\mathbb{N})$$

by iterating the *insert* algorithm of Example 5.2:

$$\mathit{isort} \coloneqq \mathbf{foldr}\{r.\textsc{CList}_1^r\ (\mathbf{F}\mathbb{N})\}[r.[]; r.(\mathbf{ret}\ x).l.\ \mathit{insert}\ x\ l]().$$

The family $B(r) \coloneqq \textsc{CList}_1^r\ (\mathbf{F}\mathbb{N})$ relies on the specification of *insert*, which is parametric in the linear credits (p in Example 5.2). The fact that this program only takes a triangular credit annotation of 1 on the input list guarantees implicitly that insertion sort has the desired cost upper bound. ⌟

5.2 A Resource-Aware Semantics

Semantically, Giralf can be defined as a *sub-language* of Calf. Every Giralf type is already present in Calf, and a program typing judgment $\Delta \vdash^q e : A$ in Giralf is interpreted as a Calf program $e : \triangleright^q(\otimes\Delta) \nrightarrow A$, moving credits via Lemmas 4.5, 4.6 and 4.12. As an invariant, this interpretation has zero cost in the abstract phase.

Example 5.4 (✋). The term $\mathbf{store}\langle p\rangle(e)$ is interpreted as the composite

$$\triangleright^q(\otimes\Delta) \xrightarrow{\text{Lemma 4.5}} \triangleright^{p+q'}(\otimes\Delta) \xrightarrow{\text{Lemma 4.6}} \triangleright^p(\triangleright^{q'}(\otimes\Delta)) \xrightarrow{\triangleright^p(e)} \triangleright^p A$$

credits as output. Then, the term $\mathbf{spend}\langle p\rangle(e)$ is interpreted as post-composition of the above chain with $\mathrm{spend}\langle p\rangle : \triangleright^p A \multimap A$. The use of $\mathrm{spend}\langle p\rangle$, *not* $\mathbf{charge}\langle p\rangle$, highlights a fundamental difference between Calf and of Giralf: although the cost effect $\mathbf{charge}\langle p\rangle : A \multimap A$ may be performed arbitrarily in Calf, cost may only be incurred in Giralf via the operation $\mathrm{spend}\langle p\rangle : \triangleright^p A \multimap A$ when the cost has already been paid for upfront. ⌟

Remark 5.5. Given a Giralf program interpreted as $e : \triangleright^q(\otimes\Delta) \nrightarrow A$, pre-paying for the q credits via the Calf program $e \circ \mathrm{save}\langle q\rangle : \otimes\Delta \nrightarrow A$ reframes the credit context q as the amortized cost publicized in the abstract phase rather than as credits attached to the input.

$$
\begin{array}{rr}
[\![A]\!] : \mathcal{V} & \Phi_A : [\![A]\!] \to \mathbb{C} \\
[\![\mathbf{F}X]\!] \coloneqq X & \Phi_{\mathbf{F}X}(x) \coloneqq 0 \\
[\![\top]\!] \coloneqq 1 & \Phi_\top() \coloneqq 0 \\
[\![A \otimes B]\!] \coloneqq [\![A]\!] \times [\![B]\!] & \Phi_{A\otimes B}(a,b) \coloneqq \Phi_A(a) + \Phi_B(b) \\
[\![A_1 + A_2]\!] \coloneqq [\![A_1]\!] + [\![A_2]\!] & \Phi_{A_1+A_2}(\mathbf{inj}_i\ a_i) \coloneqq \Phi_{A_i}(a_i) \\
[\![\triangleright^p A]\!] \coloneqq [\![A]\!] & \Phi_{\triangleright^p A}(a) \coloneqq p + \Phi_A(a) \\
[\![\textsc{CList}_1^p\ A]\!] \coloneqq \textsc{List}\ [\![A]\!] & \Phi_{\textsc{CList}_1^p\ A}(l) \coloneqq |l| \cdot p + \displaystyle\sum_{a\in l} \Phi_A(a) \\
[\![\textsc{CList}_2^{(p_1,p_2)}\ A]\!] \coloneqq \textsc{List}\ [\![A]\!] & \Phi_{\textsc{CList}_2^{(p_1,p_2)}\ A}(l) \coloneqq |l| \cdot p_1 + \binom{|l|}{2} \cdot p_2 + \displaystyle\sum_{a\in l} \Phi_A(a)
\end{array}
$$

Fig. 2. The potential-based semantics of AARA types [Hoffmann and Hofmann 2010b].

The Giralf language presented here is directly inspired by AARA [Hoffmann and Hofmann 2010b] and, when considering terminating functions, generalizes it. In AARA, adapting to the notation of this work, it is common to consider only positive[10] types, defined by a grammar such as

$$A, B, C ::= \mathbf{F}X \mid \top \mid A \otimes B \mid A + B \mid \triangleright^p A \mid \textsc{CList}_1^p\ A \mid \textsc{CList}_2^{(p_1,p_2)}\ A.$$

Note that this is an inductively-defined subset of the computation types available in Calf (and thus Giralf). In the semantics of AARA [Hoffmann and Jost 2022], each of these types is assigned a set of values $[\![A]\!] : \mathcal{V}$, and then a potential function Φ_A is defined by induction on types in Fig. 2.

Remark 5.6. In such a semantics, it is not obvious how to incorporate negative types. For example, what potential (of type $\mathbb{C}$) would a value of type $\triangleright^1\top \times \triangleright^2\top$ have—or worse, $(n : \mathbb{N}) \rightharpoonup \triangleright^n\top$?

We may build such a potential function Φ_A into a type, $\mathbf{Potential}(\Phi_A : [\![A]\!] \to \mathbb{C})$, using Definition 2.21. In fact, the potential function Φ_A is already contained within the Calf type A.

Theorem 5.7. *Let A be a type in the AARA grammar given above. Viewed as a Calf type, A is equivalent to the type $\mathbf{Potential}(\Phi_A : [\![A]\!] \to \mathbb{C})$.*

Proof Sketch. By induction on the AARA type grammar. From the physicist's view, as the type $\mathbf{Potential}(\Phi_A : [\![A]\!] \to \mathbb{C})$ is given in terms of $\mathbf{F}[\![A]\!]$ (by Definition 2.21), the cases follow by the fact that $\mathbf{F}$ (as a left adjoint) preserves positive types. Or, from the banker's view, the cases follow from Lemmas 4.6 and 4.10 and credits commuting with positive types. □

This shows not only that the potential functions Φ_A of AARA are present in Calf types, but also that the connectives used here generalize AARA in a compatible way. Then, framed in terms of Φ_A, the soundness theorem for AARA follows immediately from the Kripke semantics of Calf.

Theorem 5.8 (Soundness of AARA). *Let $\Delta \vdash^q e : A$ be a typing judgment in AARA, and suppose the program e incurs $p : \mathbb{C}$ cost upon evaluation in some environment $\delta : [\![\Delta]\!]$. Then, the specification q is a sound upper bound for p, up to the following conservation condition:*

$$p + \Phi_A(e(\delta)) \le \Phi_{\otimes\Delta}(\delta) + q.$$

[10] Due to the call-by-value nature of AARA, the function types of AARA correspond to the type $\mathbf{F}(A \multimap B)$.

Proof. View the AARA program e as a Giralf program $e : \otimes\Delta \multimap A$. By Theorem 5.7, we have

$$\Delta = \mathbf{Potential}(\Phi_{\otimes\Delta} : [\![\otimes\Delta]\!] \to \mathbb{C}) \qquad \text{and} \qquad A = \mathbf{Potential}(\Phi_A : [\![A]\!] \to \mathbb{C}).$$

Then, in the Kripke semantics of Calf, e is interpreted as the following lax commutative square:

$$\begin{array}{ccc} [\![\mathbf{F}[\![\otimes\Delta]\!]]\!] & \xrightarrow{\quad [\![e]\!]_\top \quad} & [\![\mathbf{F}[\![A]\!]]\!] \\ {\scriptstyle \varphi(\Phi_{\otimes\Delta})}\downarrow & \geq & \downarrow{\scriptstyle \varphi(\Phi_A(a))} \\ [\![\mathbf{F}[\![\otimes\Delta]\!]]\!] & \xrightarrow[\quad [\![e]\!]_{\mathrm{abs}} \circ \mathbf{charge}\langle q\rangle \quad]{} & [\![\mathbf{F}[\![A]\!]]\!] \end{array}$$

The cost portion of this square is precisely the desired conservation condition. □

Thus, Calf can be viewed as a conservative extension of AARA. When restricting attention to the AARA-like types in Giralf, the language behaves just like AARA, but Giralf supports additional types, data abstraction, and manual verification. Conversely, when Calf programs happen to lie in the Giralf sub-language, it is possible to automatically infer cost bounds.

5.3 An Inference Algorithm

In Giralf, credits are placed within data structures to ensure the availability of a credit whenever cost is incurred. Beyond streamlining the manual development of programs involving credits, this realization of the banker's view enables automated *cost inference* by linear programming as in AARA and RaML [Hoffmann et al. 2012b]. Cost inference takes a Calf program and finds an analogous program in Giralf which, by construction, assumes all cost is prepaid for upfront.

Definition 5.9 (Cost Inference). Let $e : A \multimap B$ be a Calf program. An *inference for* e is a Giralf term $a : \underline{A} \vdash^q \underline{e} : \underline{B}$ such that

(1) assuming $\neg\mathbf{abs}$, it is the case that $\underline{A} = A$, $\underline{B} = B$, and $\underline{e} = e$; and
(2) assuming $\mathbf{abs}$, it is the case that $\underline{e}$ is the cost erasure of e.

The assumption of $\neg\mathbf{abs}$ isolates only the true execution behavior of the program; in particular, this assumption erases credit annotations.

Lemma 5.10. *Assuming* $\neg\mathbf{abs}$*, it is the case that* $\mathbf{Abstraction}(\alpha : A_\top \multimap A_{\mathrm{abs}}) = A_\top$.

Corollary 5.11. *Assuming* $\neg\mathbf{abs}$*, it is the case that:*

(1) $\triangleright^p A = A$ *and* $\mathbf{spend}\langle c\rangle = \mathbf{charge}\langle c\rangle$*;*
(2) $\triangleleft^p A = A$*; and*
(3) $\textsc{CList}_1^c\ A = \textsc{CList}_2^{(c_1,c_2)}\ A = \textsc{List}\ A$.

Example 5.12. The Giralf insertion sort program of Example 5.3, annotated with $\mathbf{spend}\langle-\rangle$, is an inference for the Calf insertion sort program of Niu et al. [2022], annotated with $\mathbf{charge}\langle-\rangle$. ⌟

An *inference algorithm* is licensed to alter the types in a program to include credits. Note that inference may fail, as the costs included in an arbitrary Calf program can be arbitrarily complex [Niu et al. 2022]. Building on the linear programming-based cost inference techniques of AARA [Hofmann and Jost 2003], we now describe a cost inference algorithm for a sub-language of Calf.

5.3.1 *Skeletal Translation.*

For inference, we restrict attention to types in the following grammar:

$$A, B, C ::= \top \mid A + B \mid A \times B \mid \textsc{List}\ (\mathbf{F}X).$$

We choose these as representative cases, but it is straightforward to accommodate similar types of AARA. Inference for the simple positive types $\top$ and $A + B$ is well-understood [Hofmann and Jost

2003]; inference for the lazy product type $A \times B$ is novel; and inference for the list type $\textsc{List}\ (\mathbf{F}X)$ is involved when quadratic credits are considered due to the requirement of structural recursion.

Inference begins by inductively defining $\underline{A} : C$, an augmentation of A with the structure of credits and debits that leaves the precise numbers yet unspecified (indicated by a ? symbol).

$$\begin{aligned}
\underline{A} &: C \\
\underline{\top} &\coloneqq \top \\
\underline{A_1 + A_2} &\coloneqq \triangleright^? \underline{A_1} + \triangleright^? \underline{A_2} \\
\underline{A_1 \times A_2} &\coloneqq \triangleleft^? \underline{A_1} \times \triangleleft^? \underline{A_2} \\
\underline{\textsc{List}\ (\mathbf{F}X)} &\coloneqq \textsc{CList}_2^{(?,?)}\ (\mathbf{F}X)
\end{aligned}$$

Note that $\neg\mathbf{abs}\!\rightarrow\!(\underline{A} = A)$ by Corollary 5.11. The duality of sums and products appears via the duality of credits and debits: when eliminating from a sum/introducing a product, each case/component may require a different amount of credits. We annotate lists with linear and quadratic credits.

On terms, inference augments Calf programs $e : A \multimap B$ to Giralf programs $\underline{e}$ such that $\underline{A} \vdash^? \underline{e} : \underline{B}$. For example, the cost effect is reframed as spending credits, and the sum and product cases follow routinely from the type translations.

$$\begin{aligned}
\underline{\mathbf{charge}\langle c\rangle(e)} &\coloneqq \mathbf{spend}\langle c\rangle(\underline{e}) \\
\underline{\mathbf{inj}_i\ e} &\coloneqq \mathbf{inj}_i\ (\mathbf{store}\langle ?\rangle(\underline{e})) \\
\underline{\mathbf{case}(e; a_1.e_1; a_2.e_2)} &\coloneqq \mathbf{case}(e; a_1'.\mathbf{let\ store}(a_1) = a_1'\ \mathbf{in}\ \underline{e_1}; a_2'.\mathbf{let\ store}(a_2) = a_2'\ \mathbf{in}\ \underline{e_2}) \\
\underline{(e_1, e_2)} &\coloneqq (\mathbf{get}\langle ?\rangle(\underline{e_1}), \mathbf{get}\langle ?\rangle(\underline{e_2})) \\
\underline{\mathbf{proj}_i\ e} &\coloneqq \mathbf{pay}(\mathbf{proj}_i\ \underline{e})
\end{aligned}$$

The recursion principle for the list recursor is more involved, due to the translation of lists as quadratic-credit lists. Consider the recursor $\mathbf{foldr}[e_0; a.b.e_1](e) : B_0$. Although we could naively translate to a recursor for quadratic-credit lists at the constant family $\underline{B}(r) \coloneqq \underline{B_0}$, there are two issues that arise in common examples. Recall the typing judgment for quadratic-credit lists (Section 5.1.4).

(1) In the base case e_0, no credits are made available, even though cost could be incurred in the original Calf program. To mitigate this issue, it is common to thread credits through using the debit operator, similar to the credit-passing aspect of Lemma 5.1.
(2) In the inductive case e_1, some unknown quantity r of credits is made available. However, if any of that is to be spent, it must be known that r is large enough. For this reason, it is important to build an *invariant* into the output family that restricts the possible values of r, representable using a power type $? \rightharpoonup \cdots$.

We solve both of these issues by choosing a family $\underline{B}(r) \coloneqq ? \rightharpoonup \triangleleft^? \underline{B_0}$ (where both the unknown invariant ? and the unknown debit annotation ? may depend on r) and define

$$\underline{\mathbf{foldr}[e_0; x.a.e_1](e)} \coloneqq \mathbf{pay}(\mathbf{foldr}\{r.\underline{B}(r)\}[r.\underline{e_0}'; r.a.b.\underline{e_1}'](\underline{e})\ ?)$$

where $\underline{e_0}' \coloneqq \lambda\ (h : ?) \rightharpoonup \mathbf{get}\langle ?\rangle(\underline{e_0})$ and $\underline{e_1}' \coloneqq \lambda\ (h : ?) \rightharpoonup \mathbf{get}\langle ?\rangle([\mathbf{pay}(b\ ?)/b]\underline{e_1})$ thread the debits and invariant recursively through the code. To correctly close the loop, the complementary elimination forms are then be applied to the recursive result b and the entire **foldr** expression.

Example 5.13. As a basic example, consider *snoc*, a worst-case simplification of *insert* (Example 5.2) that appends an element $x : \mathbb{N}$ to the end of a list:

$$\begin{aligned}
&\mathit{snoc} : \textsc{List}\ (\mathbf{F}\mathbb{N}) \multimap \textsc{List}\ (\mathbf{F}\mathbb{N}) \\
&\mathit{snoc}\ l \coloneqq \mathbf{foldr}[\mathbf{ret}\ x :: []; (\mathbf{ret}\ y).b.\mathbf{charge}\langle 1\rangle(\mathbf{ret}\ y :: b)](l)
\end{aligned}$$

This program translates to the following skeletal Giralf program:

$$\begin{aligned}
&\underline{snoc} : \textsc{List}_2^{(?,?)}\ (\mathbb{FN}) \multimap \textsc{List}_2^{(?,?)}\ (\mathbb{FN})\\
&\underline{snoc}\ l \coloneqq \mathbf{pay}(\underline{snoc'}\ ?)\ \mathbf{where}\\
&\quad e_0 \coloneqq \lambda\ (h : ?) \multimap \mathbf{get}\langle ? \rangle(\mathbf{ret}\ x :: [])\\
&\quad e_1 \coloneqq \lambda\ (h : ?) \multimap \mathbf{get}\langle ? \rangle(\mathbf{spend}\langle 1 \rangle(\mathbf{ret}\ y :: (\mathbf{pay}(b\ ?))))\\
&\quad \underline{snoc'} \coloneqq \mathbf{foldr}\{r.? \multimap \triangleleft^{?}\textsc{List}_2^{(?,?)}\ (\mathbb{FN})\}[r.e_0; r.(\mathbf{ret}\ y).b.e_1](l)
\end{aligned}$$

This skeletal Giralf code faithfully reconstructs *snoc* with the assumption that all cost is prepaid. ⌟

5.3.2 Constraint Solving. The crux of cost inference is determining the unknown amounts of credits, which will induce the final cost bound. We achieve this by adapting the LP-based approach of RaML [Hoffmann et al. 2012b; Hoffmann and Hofmann 2010a], which performs cost inference on programs with unbounded recursion, to the present setting with structural recursion.

Because RaML is implemented using an LP solver, it always outputs resource bounds with concrete numbers rather than symbolic expressions. This causes the precise technique to be incompatible with the quadratic-credit list recursor, which uses an inductive family $B(r)$ due to the fact that the linear coefficient r changes inductively. In particular, the requisite invariant must be a symbolic expression involving r. To surmount this problem, we introduce the notion of *cost-free* and *cost-aware* annotations [Hoffmann et al. 2012a]. For a skeletal Giralf program e,

(1) a *cost-aware annotation* is a valid numerical assignment of credit values to unknowns; and
(2) a *cost-free annotation* is a cost-aware annotation for $\widehat{e}$, constructed by erasing all cost annotations in e, describing only how credits move from input to output.

Example 5.14. Let us continue the running example of *snoc*. To solve for its unknown credit values, we run the RaML algorithm on the recursor $\underline{snoc'}$:

(1) One *cost-aware* annotation demonstrates that $\underline{snoc'}$ spends at most $|l|$ credits:

$$l : \textsc{List}_2^{(1,0)}\ (\mathbb{FN}) \vdash^0 \underline{snoc'} : ? \multimap \triangleleft^0\textsc{List}_2^{(0,0)}\ (\mathbb{FN}).$$

(2) One *cost-free* annotation describes the movement of linear credits:

$$l : \textsc{List}_2^{(1,0)}\ (\mathbb{FN}) \vdash^0 \widehat{\underline{snoc'}} : ? \multimap \triangleleft^1\textsc{List}_2^{(1,0)}\ (\mathbb{FN}).$$

Given input list l carrying $|l|$ credits, $\underline{snoc'}$ outputs a list l' carrying $|l'|$ credits, but at the additional cost of 1 more credit (indicated by the debit operator) because $|l'| = |l| + 1$.

(3) Another *cost-free* annotation describes the movement of quadratic credits:

$$l : \textsc{List}_2^{(1,1)}\ (\mathbb{FN}) \vdash^0 \widehat{\underline{snoc'}} : ? \multimap \triangleleft^0\textsc{List}_2^{(0,1)}\ (\mathbb{FN}).$$

RaML is capable of automating all three of the inferences shown above. ⌟

Numerical cost-aware and cost-free types can be combined into a symbolic invariant based on the following two lemmas [Hoffmann and Hofmann 2010a].

Lemma 5.15. Cost-free annotations *are closed under addition and scalar multiplication. That is, if* $\Delta_1 \vdash^{q_1} \widehat{\underline{e}} : A_1$ *and* $\Delta_2 \vdash^{q_2} \widehat{\underline{e}} : A_2$ *are credit assignments on the same skeletal Giralf types, then*

$$\Delta_1 \oplus \Delta_2 \vdash^{q_1+q_2} \widehat{\underline{e}} : A_1 \oplus A_2 \qquad \textit{and} \qquad k \odot \Delta_1 \vdash^{k\odot q_1} \widehat{\underline{e}} : k \odot A_1$$

where $\oplus$ *and* $\odot$ *are pointwise addition and scalar multiplication of the assigned credit values.*

Lemma 5.16. Cost-aware annotations *are closed under addition with a* cost-free annotation *of the same term. That is, if* $\Delta_1 \vdash^{q_1} \underline{e} : A_1$ *and* $\Delta_2 \vdash^{q_2} \widehat{\underline{e}} : A_2$*, then* $\Delta_1 \oplus \Delta_2 \vdash^{q_1+q_2} \underline{e} : A_1 \oplus A_2$.

Example 5.17. With these lemmas, we combine the numerical annotations of Example 5.14 into a single type with symbolic credit variables. Denoting the three types as A_{ca}, A_{cf1}, and A_{cf2}, we combine them via $A_{\text{ca}} \oplus (p_1 \odot A_{\text{cf1}}) \oplus (p_2 \odot A_{\text{cf2}})$ to obtain the desired resource-polymorphic typing

$$l : \textsc{CList}_2^{(1+p_1+p_2,p_2)}\ (\mathbf{F}\mathbb{N}) \vdash^0 \underline{\mathit{snoc}'} : ? \multimap \triangleleft^{p_1}\textsc{CList}_2^{(p_1,p_2)}\ (\mathbf{F}\mathbb{N}).$$

Then, with a change of basis $r \coloneqq 1 + p_1 + p_2$, we determine the full type family for $\underline{\mathit{snoc}'}$:

$$B(r) \coloneqq (r \geq 1 + p_2) \multimap \triangleleft^{r-p_2-1}\textsc{CList}_2^{(r-p_2-1,p_2)}\ (\mathbf{F}\mathbb{N}).$$

The invariant predicate guarantees that the changing linear coefficient r is always large enough by ensuring all subtractions of credits are non-negative. ⌟

In summary, using the LP solving technique of AARA and RaML, we infer valid invariants and credit amounts. Once the unknown credit amounts within the skeletal Giralf *types* are solved, we populate the unknown credit amounts within the skeletal Giralf *terms* via a simple algorithm inspired by bidirectional type checking; *e.g.*, when checking $\mathbf{get}\langle ? \rangle(e)$ against the type $\triangleleft^p A$, we replace ? with p. The only remaining missing data are proofs that the symbolic invariants are recursively preserved; we generate such proofs using a simple heuristic-driven arithmetic solver.

5.3.3 Cost Inference for Calf. Using the above inference procedure and the semantics of Giralf in Calf, we may infer cost bounds on Calf programs automatically.

Example 5.18. Let $\mathit{isort} : \mathbf{U}(\textsc{List}\ \mathbb{N} \multimap \mathbf{F}(\textsc{List}\ \mathbb{N}))$ be the insertion sort algorithm in Calf [Niu et al. 2022]. This program may be written with the equivalent types $\mathit{isort} : \mathbf{F}(\textsc{List}\ \mathbb{N}) \multimap \mathbf{F}(\textsc{List}\ \mathbb{N})$ and $\mathit{isort} : \textsc{List}\ (\mathbf{F}\mathbb{N}) \multimap \textsc{List}\ (\mathbf{F}\mathbb{N})$. The previously described inference algorithm derives the judgment

$$\textsc{CList}_2^{(p_1,p_2+1)}\ (\mathbf{F}\mathbb{N}) \vdash^0 \underline{\mathit{isort}} : \textsc{CList}_2^{(p_1,p_2)}\ (\mathbf{F}\mathbb{N})$$

demonstrating that the quadratic credits stored alongside the input list decrease by 1, indicating a triangular cost upper bound. Let $p_1 = p_2 = 0$; as a Calf program (up to Lemma 4.6),

$$\underline{\mathit{isort}} : \textsc{CList}_2^{(0,1)}\ (\mathbf{F}\mathbb{N}) \multimap \mathbf{F}(\textsc{List}\ \mathbb{N}),$$

and by Theorem 5.7 and Lemma 4.10, with $\Phi(l) \coloneqq \binom{|l|}{2}$ we have that

$$\underline{\mathit{isort}} : (l : \textsc{List}\ \mathbb{N}) \rtimes \rhd^{\Phi(l)} \top \multimap \mathbf{F}(\textsc{List}\ \mathbb{N}).$$

So, pre-paying as in Remark 5.5, we find by Lemma 4.8 that

$$\lambda\ (\mathbf{ret}\ l) \multimap \underline{\mathit{isort}}(l, \mathsf{save}\langle \Phi(l) \rangle) : \mathbf{F}(\textsc{List}\ \mathbb{N}) \multimap \mathbf{F}(\textsc{List}\ \mathbb{N})$$

is an upper bound for the original *isort* program in Calf in the abstract phase. ⌟

6 RELATED WORK

The central ideas on which the present work is based are verified cost analysis in Calf [Grodin and Harper 2024; Grodin et al. 2024; Niu et al. 2022], fracture and gluing for synthetic abstraction [Grodin et al. 2026; Rijke et al. 2020], and AARA [Das et al. 2021; Hoffmann and Jost 2022; Hofmann and Jost 2003]. However, there are many other approaches to formally verifying amortized cost.

Using potential functions from the physicist's view, van Brügge [2024] and Nipkow and Brinkop [2019] mechanize various amortized bounds. Atkey [2014] develops a connection between abstraction and conservation of energy which parallels the perspective taken in this work.

Cutler et al. [2020] propose a non-dependent graded language similar to Giralf whose semantics is also based on the cost effect (a writer monad). Danielsson [2008] verifies amortized costs in dependent type theory using a graded monad representing debits, and Atkey [2011], Mével et al. [2019], and Pottier et al. [2024] extend separation logics with a credit resource.

λ-amor. Rajani et al. [2024] present λ-amor, an extension of the linear type theory of AARA with a layer of structural refinements, roughly corresponding to the layer of value types in Giralf. Although Giralf and λ-amor have some key technical differences—λ-amor includes, for example, unbounded recursion and impredicative quantification—the overall structures and goals of the type theories are aligned. Notably, λ-amor is the first work based on the banker's method of which we are aware that includes negative types, such as (in our notation) products $A \times B$, powers $X \multimap A$, homomorphisms $A \multimap B$, and debits $\triangleleft^c A$. Semantically, this is achieved by extending the potential functions of AARA (reviewed in Section 5.2) to potential *predicates* $\Phi_A : [\![A]\!] \to \mathbb{C} \to \mathsf{Prop}$ that determine if a given amount of potential is sufficient to construct a particular value. Glimmers of the lax commutative squares emphasized in the present work are visible in these predicates.

Dependent AARA. More recently, Xu and Wang [2026] extended AARA to support potential that is explicitly dependent on data, written (adapting our notation) $\Gamma ; \Phi_\Gamma \vdash e : X ; \Phi_X$, where $\Phi_\Gamma : \Gamma \to \mathbb{C}$ and $\Phi_X : X \to \mathbb{C}$ describe the input and output potentials. We conjecture that their lightweight dependent type system can be understood in our full dependent type theory by interpreting their typing judgment as the type of homomorphisms $\mathbf{Potential}(\Phi_\Gamma : \Gamma \to \mathbb{C}) \multimap \mathbf{Potential}(\Phi_X : X \to \mathbb{C})$.

7 CONCLUSION

In this work, we render amortized analysis synthetically in the Calf dependent type theory, treating the cost of an effectful abstraction function built into a type as potential. This ensures that every program comes equipped with an amortized cost interface coherent up to a generalization of the physicist's conservation of potential that accounts for both cost and behavior. Defining the banker's credits and debits synthetically, we develop Giralf, a substructural dependent type theory layered on top of Calf streamlining programming with credits. Extending the automated cost inference techniques of AARA, we present an automated cost inference procedure for Calf targeting Giralf.

Future Work. In this work, we focus on amortization for ephemeral data structures. Okasaki [1999] uses laziness to adapt amortization to the persistent setting, developed logically by Pottier et al. [2024] and operationally by Lorenzen [2026]. It is yet unknown if the technique developed in the present work can be adapted to the setting of persistent amortization.

In this work, we made use of a commutative cost model, which was essential for the semantics of types such as tensor products and credits. However, in AARA, a non-commutative cost model is sometimes used to study high-water marks on cost [Hoffmann and Hofmann 2010a]. We leave the incorporation of such cost models to future work.

The methods of cost inference in AARA have been developed extensively over decades [Hoffmann and Jost 2022]. In this work, we adapt the inference algorithm for polynomial credits on lists [Hoffmann and Hofmann 2010b], but we leave it as future work to integrate features such as multivariate bounds [Hoffmann et al. 2011, 2012a], exponential bounds [Kahn and Hoffmann 2020], and more general inductive types [Grosen et al. 2023].

DATA AVAILABILITY STATEMENT

The core ideas presented in this work are implemented and mechanized. Fusing and extending the mechanizations of Niu et al. [2022], Grodin et al. [2024], and Grodin et al. [2026], we provide a mechanization in Cubical Agda [Norell 2009; Vezzosi et al. 2019] (indicated by 🤌) of the central constructions and theorems of Sections 2 to 4 and the semantics of Giralf described in Section 5. As an approximation, the inequality structure of Grodin et al. [2024] is replaced with equality.

We also provide an implementation of the inference algorithm given in Section 5.3, adapting the OCaml code of RaML [Hoffmann et al. 2012b]; while this code is not verified, it is *certifying*, emitting Giralf artifacts in Agda that include a certificate of amortized correctness by construction.

ACKNOWLEDGMENTS

The authors thank Reid Barton for his advice on the proof of Theorem 2.15 and Jonathan Sterling for fruitful adjacent collaboration.

This material is based upon work supported by Jane Street Group, LLC; the United States Air Force Office of Scientific Research under grant numbers FA9550-21-0009 and FA9550-23-1-0434 (Tristan Nguyen, program manager); and the National Science Foundation under award numbers 2615896, 2311983, and 2525102. 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 and the NSF.

REFERENCES

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

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science* Volume 7, Issue 2 (June 2011). https://doi.org/10.2168/LMCS-7(2:17)2011

Robert Atkey. 2014. From Parametricity to Conservation Laws, via Noether's Theorem. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. Association for Computing Machinery, New York, NY, USA, 491–502. https://doi.org/10.1145/2535838.2535867

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

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

Joseph W. Cutler, Daniel R. Licata, and Norman Danner. 2020. Denotational Recurrence Extraction for Amortized Analysis. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 97:1–97:29. https://doi.org/10.1145/3408979

Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. *ACM SIGPLAN Notices* 43, 1 (Jan. 2008), 133–144. https://doi.org/10.1145/1328897.1328457

Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2021. Resource-Aware Session Types for Digital Contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–16. https://doi.org/10.1109/CSF51468.2021.00004

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

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

Marco Grandis and Robert Paré. 2004. Adjoint for Double Categories. *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 45, 3 (2004), 193–240. https://www.numdam.org/item/?id=CTGDC_2004__45_3_193_0

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

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

Harrison Grodin, Runming Li, and Robert Harper. 2026. Abstraction Functions as Types: Modular Verification of Cost and Behavior in Dependent Type Theory. *Proceedings of the ACM on Programming Languages* 10, POPL (Jan. 2026), 31:895–31:922. https://doi.org/10.1145/3776673

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

Jessie Grosen, David M. Kahn, and Jan Hoffmann. 2023. Automatic Amortized Resource Analysis with Regular Recursive Types. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–14. https://doi.org/10.1109/LICS56636.2023.10175720

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

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

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. Association for Computing Machinery, New York, NY, USA, 357–370. https://doi.org/10.1145/1926385.1926427

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012a. Multivariate Amortized Resource Analysis. *ACM Transactions on Programming Languages and Systems* 34, 3 (Nov. 2012), 14:1–14:62. https://doi.org/10.1145/2362389.2362393

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012b. Resource Aware ML. In *Computer Aided Verification (Lecture Notes in Computer Science)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, Berlin, Heidelberg, 781–786. https://doi.org/10.1007/978-3-642-31424-7_64

Jan Hoffmann and Martin Hofmann. 2010a. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Kazunori Ueda (Ed.). Springer, Berlin, Heidelberg, 172–187. https://doi.org/10.1007/978-3-642-17164-2_13

Jan Hoffmann and Martin Hofmann. 2010b. Amortized Resource Analysis with Polynomial Potential. In *Programming Languages and Systems*, Andrew D. Gordon (Ed.). Springer, Berlin, Heidelberg, 287–306. https://doi.org/10.1007/978-3-642-11957-6_16

Jan Hoffmann and Steffen Jost. 2022. Two Decades of Automatic Amortized Resource Analysis. *Mathematical Structures in Computer Science* 32, 6 (June 2022), 729–759. https://doi.org/10.1017/S0960129521000487

Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. *ACM SIGPLAN Notices* 38, 1 (Jan. 2003), 185–197. https://doi.org/10.1145/640128.604148

Martin Hofmann, Lorenz Leutgeb, David Obwaller, Georg Moser, and Florian Zuleger. 2022. Type-Based Analysis of Logarithmic Amortised Complexity. *Mathematical Structures in Computer Science* 32, 6 (June 2022), 794–826. https://doi.org/10.1017/S0960129521000232

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

David M. Kahn and Jan Hoffmann. 2020. Exponential Automatic Amortized Resource Analysis. In *Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer International Publishing, Cham, 359–380. https://doi.org/10.1007/978-3-030-45231-5_19

Lukas Kebuladze. 2025. *Formally Verified Amortized Cost Analysis of Splay Trees in Agda.* Technical Report. Carnegie Mellon University. https://www.cs.cmu.edu/~rwh/code/kebuladze_splay_tree.tar.gz

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

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

Runming Li and Robert Harper. 2025. Canonicity for Cost-Aware Logical Framework via Synthetic Tait Computability. https://doi.org/10.48550/arXiv.2504.12464 arXiv:2504.12464 [cs]

Anton Lorenzen. 2026. Persistent Amortised Analysis, Operationally. https://doi.org/arXiv:2605.09411

Lambert Meertens. 1992. Paramorphisms. *Formal Aspects of Computing* 4, 5 (Sept. 1992), 413–424. https://doi.org/10.1007/BF01211391

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Tobias Nipkow and Hauke Brinkop. 2019. Amortized Complexity Verified. *Journal of Automated Reasoning* 62, 3 (March 2019), 367–391. https://doi.org/10.1007/s10817-018-9459-3

Yue Niu and Robert Harper. 2022. A Metalanguage for Cost-Aware Denotational Semantics. https://doi.org/10.48550/arXiv.2209.12669 arXiv:2209.12669 [cs]

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

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

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

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

Long Pham, Yue Niu, Nathan Glover, Feras Saad, and Jan Hoffmann. 2025. Integrating Resource Analyses via Resource Decomposition. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (Oct. 2025), 409:3811–409:3840. https://doi.org/10.1145/3763798

François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2024. Thunks and Debits in Separation Logic with Time Credits. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 50:1482–50:1508. https://doi.org/10.1145/3632892

Vineet Rajani. 2020. *A Type-Theory for Higher-Order Amortized Analysis.* doctoralThesis. Saarländische Universitäts- und Landesbibliothek. https://doi.org/10.22028/D291-30877

Vineet Rajani, Gilles Barthe, and Deepak Garg. 2024. A Modal Type Theory of Expected Cost in Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 8, OOPSLA2 (Oct. 2024), 285:389–285:414. https://doi.org/10.1145/3689725

Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2021. A Unifying Type-Theory for Higher-Order (Amortized) Cost Analysis. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 27:1–27:28. https://doi.org/10.1145/3434308

John C. Reynolds. 1983. Types, Abstraction, and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523. https://doi.org/10.1007/3-540-55511-0_1

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

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

Daniel D. Sleator and Robert E. Tarjan. 1985a. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM* 28, 2 (Feb. 1985), 202–208. https://doi.org/10.1145/2786.2793

Daniel Dominic Sleator and Robert Endre Tarjan. 1985b. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (July 1985), 652–686. https://doi.org/10.1145/3828.3835

Ross Street. 1974. Fibrations and Yoneda's Lemma in a 2-Category. In *Category Seminar*, Gregory M. Kelly (Ed.). Springer, Berlin, Heidelberg, 104–133. https://doi.org/10.1007/BFb0063102

Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (April 1985), 306–318. https://doi.org/10.1137/0606031

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

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

Jan van Brügge. 2024. Liquid Amortization: Proving Amortized Complexity with LiquidHaskell (Functional Pearl). In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium (Haskell 2024).* Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/3677999.3678282

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proceedings of the ACM on Programming Languages* 3, ICFP (July 2019), 87:1–87:29. https://doi.org/10.1145/3341691

Han Xu and Di Wang. 2026. Dependently-Typed AARA: A Non-Affine Approach for Resource Analysis of Higher-Order Programs. In *Programming Languages and Systems*, Robbert Krebbers (Ed.). Springer Nature Switzerland, Cham, 362–391. https://doi.org/10.1007/978-3-032-22723-2_13