A Cost-Aware Logical Framework

YUE NIU, Carnegie Mellon University, USA
JONATHAN STERLING, Aarhus University, DK
HARRISON GRODIN, Carnegie Mellon University, USA
ROBERT HARPER, Carnegie Mellon University, USA

We present **calf**, a **c**ost-**a**ware **l**ogical **f**ramework for studying quantitative aspects of functional programs. Taking inspiration from recent work that reconstructs traditional aspects of programming languages in terms of a modal account of *phase distinctions*, we argue that the cost structure of programs motivates a phase distinction between *intension* and *extension*. Armed with this technology, we contribute a synthetic account of cost structure as a computational effect in which cost-aware programs enjoy an internal noninterference property: input/output behavior cannot depend on cost. As a full-spectrum dependent type theory, **calf** presents a unified language for programming and specification of both cost and behavior that can be integrated smoothly with existing mathematical libraries available in type theoretic proof assistants.

We evaluate **calf** as a general framework for cost analysis by implementing two fundamental techniques for algorithm analysis: the *method of recurrence relations* and *physicist's method for amortized analysis*. We deploy these techniques on a variety of case studies: we prove a tight, closed bound for Euclid's algorithm, verify the amortized complexity of batched queues, and derive tight, closed bounds for the sequential and *parallel* complexity of merge sort, all fully mechanized in the Agda proof assistant. Lastly we substantiate the soundness of quantitative reasoning in **calf** by means of a model construction.

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

Additional Key Words and Phrases: algorithm analysis, cost models, phase distinction, noninterference, intensional property, behavioral verification, equational reasoning, modal type theory, mechanized proof, proof assistants, recurrence relations, amortized analysis, parallel algorithms

ACM Reference Format:
Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A Cost-Aware Logical Framework. *Proc. ACM Program. Lang.* 6, POPL, Article 9 (January 2022), 31 pages. https://doi.org/10.1145/3498670

1 INTRODUCTION

Resource usage is an important *intensional* property of programs. With a rich enough type system, extensional properties of programs can be investigated in the same language as the program is written — an approach to verification that has seen much application in type theoretic tools such as Nuprl, Coq, Agda, and Idris [Brady 2013; Constable et al. 1986; Coq Development Team 2016; Norell 2009]. Intensional properties such as cost are not typically amenable to such an internal analysis, in essence because one cannot conventionally have a function cost : bool → nat that computes

Authors' addresses: Yue Niu, yuen@andrew.cmu.edu, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA; Jonathan Sterling, jsterling@cs.au.dk, Aarhus University, Aabogade 34, Aarhus C, 8000, DK; Harrison Grodin, hgrodin@andrew.cmu.edu, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA; Robert Harper, rwh@cs.cmu.edu, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA.

2475-1421/2022/1-ART9
https://doi.org/10.1145/3498670

the cost of its input (such a "function" could not respect β-equivalence). To address this problem, one could instrument programs with their cost, but this instrumentation must not be allowed to interfere with the input/output behavior of programs.

A logical framework for cost. We contribute **calf**, a **c**ost-**a**ware **l**ogical **f**ramework for studying quantitative aspects of functional programs, combining recent work on cost recurrence extraction [Kavvos et al. 2019] and the call-by-push-value decomposition of effects in dependent type theory [Pédrot and Tabareau 2019] with recent modal account of phase distinctions and noninterference of Sterling and Harper [2021]. **calf** evinces a phase distinction between extensional and intensional aspects of code (analogous to the static–dynamic phase distinction of ML languages); then the incurrence of *cost* is treated as a computational effect that has force only in the intensional fragment, ensuring that the extensional behavior of a program does not depend on the costs of its arguments. In particular **calf** ensures that one cannot write a function whose *extension* depends on the cost component of its input.

Evaluation and implementation. We evaluate the efficacy of **calf** by formulating two widely used algorithm analysis techniques – the method of recurrence relations and the physicist's method for amortized analysis – and deploying them on a variety of case studies. We have also developed an implementation of **calf** in the Agda proof assistant.[1] The following results highlight the central contributions of our case studies, all *fully mechanized* in the Agda proof assistant:

(1) We prove an asymptotically tight and closed upper bound on the number of primitive arithmetic operations used in Euclid's algorithm for gcd.
(2) We present an amortized analysis of the cost of sequences of operations on batched queues.
(3) We prove asymptotically tight and closed upper bounds on both the sequential and *parallel* complexity of insertion sort and merge sort under the comparison cost model.

It is worth emphasizing that the presented case studies all require nontrivial mathematical reasoning, which usually presents a significant hurdle for fledgling implementations of type theories that do not come equipped with the vast number of the necessary but well-known theorems. Our implementation of **calf** alleviates this pain point by allowing one to directly *import* data types from Agda whenever they are required for an algorithm. At a high level, this design evinces an embedding of the Agda universe of "pure data types" into the effectful metalanguage of **calf** that enables one to take advantage of Agda's well-developed mathematical library.

Notation. In this paper we display all mechanized theorems as defined in the implementation using the typewriter font, *e.g.* `Calf.Types.Bounded.bound/relax`.

Metatheory and soundness. In order to be used to study the cost of programs it is important that **calf** not derive an equivalence between two programs $M, N : \mathsf{bool}$ that take a different number of steps to compute. We verify by means of a model construction that **calf** does *not* identify computations that incur different numbers of steps, the first step toward a stronger adequacy theorem that would establish the equivalence of **calf**-encodings with traditional operational cost dynamics à la Blelloch and Greiner [1995].

Parallel complexity. **calf** is compatible with many interpretations of the cost structure of programs, among which is the *cost graph* that encodes the *work* (sequential cost) and *span* (parallel cost) of a program. Thus **calf** also supports reasoning about the *parallel* complexity of programs through an equational presentation of the profiling semantics of Blelloch and Greiner [Blelloch and Greiner 1995]. By focusing on the verification of functional programs, we position **calf** to take advantage of

[1] Available at https://github.com/jonsterling/agda-calf

the elegant theory of language-based parallelism [Blelloch and Greiner 1996; Greiner and Blelloch 1999; Spoonhower et al. 2008] developed over the past three decades without descending into the space of imperative, concurrent programs in which the analogous notions are much more complex.

1.1 Synthetic Cost Analysis via Computational Effects

Although many cost verification frameworks work with the deep embedding of an object language in an ambient type theory, we take a synthetic approach by defining **calf**, a full-spectrum dependent type theory in which cost is implemented as a primitive *effect*. This view of cost is inspired by Kavvos et al. [2019], who define and extract recurrence relations of functional programs representing their (high-order) cost structure.

At first glance, cost might seem like a uniform concept that can be applied indiscriminately to any computation. This first-order view quickly falls part when we consider the costs of functions, which should be functions themselves. The question then is to introduce cost into the type theory in such a way that it can flow through the type structure compositionally. The insight of Kavvos et al. [2019] is to consider the call-by-push-value (CBPV) structure induced by a certain *cost monad*'s Eilenberg–Moore category [Levy 2004], leveraging the fine-grained type structure of CBPV to assign a compositional meaning to cost at higher type.

A cost monad is just the writer monad $\mathbb{C} \times -$ for a given monoid $(\mathbb{C}, 0, +)$; in call-by-push-value, we may interpret a *value type* by a set and a *computation type* by an algebra for $\mathbb{C} \times -$. There is a free-forgetful adjunction $\mathsf{F} \dashv \mathsf{U}$, in which the right adjoint projects the carrier set of an algebra and the left adjoint takes a set A to $\mathbb{C} \times A$; both adjoints are internalized as type constructors in CBPV. In particular, given a value type A, we can form the computation type $\mathsf{F}(A)$ whose interpretation is the free algebra. Hence in **calf**, $\mathsf{F}(A)$ classifies *free* computations of A where the costs of a sequence of computations are aggregated using the monoid structure $\mathbb{C}$, and a value $a : A$ is injected into $\mathsf{F}(A)$ via $\mathsf{ret}(a)$ as the computation yielding a incurring zero cost.

The semantic situation of the cost monad inspires a CBPV language containing a single computational effect $\mathsf{step}^c(M)$ that incurs a given cost $c : \mathbb{C}$ before computing M, such that $\mathsf{step}^c(\mathsf{step}^d(M)) \equiv \mathsf{step}^{c+d}(M)$. Indeed, **calf** is a dependently typed version of this CBPV language, defined in the style of Pédrot and Tabareau's **∂cbpv** calculus. The memorable slogan of Levy for CBPV states that "a value is, a computation does," which continues to hold in **calf**: a value *is* with no associated cost, a computation *does* using some cost.

1.2 A New Phase Distinction: Behavior *vs.* Cost

The original phase distinction between static (compile-time) and dynamic (run-time) code arose in the study of module systems [Harper et al. 1990], where light-weight static compatibility is used to facilitate the composition of modules. The idea was to disallow type-level dependence on dynamic parts of a module. Recall that a signature of a module consists of declarations of kinds of constructors (static entities) and types of expressions (dynamic entities), and a module itself consists of constructors and expressions. In the case of ML modules the phase distinction associates to every module functor a function between their static parts (kinds); in this sense, the static part of a module is entirely independent of the dynamic parts of the modules it is linked with.

In our setting a different but entirely analogous phase distinction emerges between extension/behavior and intension/cost. Every type A in **calf** can be thought of as having two parts: an intensional part $\bullet A$ characterizing its cost and an extensional part $\bigcirc A$ characterizing its extensional behavior. We say that a type is (extensional, intensional) if it is isomorphic to its (extensional, intensional) part. The phase distinction ensures that the extensional part of a program is independent of the intensional parts of its arguments. Put another way, the phase distinction of behavior and cost constitutes a *noninterference* property of intension and extension:

Noninterference. Any function $\bullet A \to \bigcirc B$ from an intensional type to an extensional type is internally equal to a constant function.

1.3 The Language of Phase Distinctions

In **calf** the phase distinction between extension and intension is achieved by adding a new abstract proposition $\P_{\mathsf{E}}$ called the "extensional phase". Whenever an assumption of type $\P_{\mathsf{E}}$ is present in the context, the cost structure of programs is rendered trivial; one can think of the fragment of **calf** where $\P_{\mathsf{E}}$ is always in the context as a version of ordinary dependent type theory in which cost is not tracked. Therefore, the extensional part of a type A can be recovered as the function space $\bigcirc A \coloneqq (\P_{\mathsf{E}} \to A)$. This extensional modality can be used to state equations between programs that have different costs but identical input-output behaviors; for instance, we can prove $\bigcirc(\mathit{insertionSort} = \mathit{mergeSort})$, even thought these algorithms have different costs under the comparison cost model for sorting. Indeed, the soundness of **calf** implies that this equation does not hold outside of $\bigcirc$.

Cost structure as proof-relevance. As we have pointed out, it makes little sense to think of cost as a property of an ordinary program, because two such programs may be equal and yet "have" different costs. On the other hand we may view cost as a *structure* (proof-relevant property) over a program, and the projection of ordinary programs from cost-instrumented programs is implemented in our setting by the unit of the extensional modality $A \to \bigcirc A$. The perspective of cost as structure is an instance of a more general phenomenon pervading present-day work in type theory: notions that are ill-posed as properties of equivalence classes of typed terms can be recovered more objectively as structures defined over equivalence classes of typed terms, as in the work of Altenkirch and Kaposi [2016a]; Coquand [2019]; Sterling and Angiuli [2021]; Sterling and Harper [2021].

1.4 Quantitative Reasoning in calf

The fundamental advantage of **calf** is that it provides a purely *equational* approach to quantitative reasoning: a useful *bound* can be placed on the number of steps engendered in a computation by equating it to another computation in which the quantity can be observed directly. For example, consider a computation $e : \mathsf{F}(A)$; if we can prove that $e = \mathsf{step}^c(\mathsf{ret}(a))$ for some value $a : A$, then we are justified to say that e has cost c. This *cost refinement* is captured by the following **calf** type:

$$\mathsf{hasCost}(A, e, c) \coloneqq \Sigma a : A.\, e =_{\mathsf{F}(A)} \mathsf{step}^c(\mathsf{ret}(a))$$

In Section 3 we consider more sophisticated refinements that express *cost bounds* rather than precise costs of computations.

There are two things to note in this definition. First, we can see that cost refinements are not primitive in **calf**; rather **calf** is a logical framework for *defining* quantitative properties such as hasCost and then *proving* refinement lemmas about those properties. Secondly, our formulation of cost bounds is only meaningful insofar as stepping is nondegenerate, *i.e.* $\nvdash \mathsf{step}^c(\mathsf{ret}(a)) = \mathsf{ret}(a)$ for any value a and nontrivial cost c. In fact this nondegeneracy property constitutes one of the soundness criteria for quantitative reasoning in **calf**, which we prove in Section 5.

Under this regime, one proves more refinements as the need arises in a verification problem or when new forms of computations are introduced. In Section 3 we present syntax-directed quantitative refinement lemmas that decompose the bounds on the cost of a computation into bounds on the costs of its constituent subcomputations.

$$
\begin{array}{r}
\textit{sig} \\
\mathsf{G} : \mathsf{Type} \\
\mathsf{n} : G \to \mathbb{N} \\
\mathsf{insertEdge} : \mathsf{edge} \to (G \xrightarrow{\log \circ \mathsf{n}} G) \\
\mathsf{isEdge} : \mathsf{edge} \to (G \xrightarrow{\log \circ \mathsf{n}} \mathsf{bool})
\end{array}
\qquad
\begin{array}{r}
\textit{sig} \\
\mathsf{G} : \mathsf{Type} \\
\mathsf{n} : G \to \mathbb{N} \\
\mathsf{insertEdge} : \mathsf{edge} \to (G \xrightarrow{\mathsf{n}} G) \\
\mathsf{isEdge} : \mathsf{edge} \to (G \xrightarrow{\lambda\_.\,1} \mathsf{bool})
\end{array}
$$

Fig. 1. Cost signatures; left is Alice and right is Bob. For simplicity, suppose the vertices are natural numbers and define edges as ordered pairs (*i.e.* $\mathsf{edge} := \mathbb{N}^2$).

1.5 Compositional Cost Analysis

As a type theory, **calf** naturally supports a compositional style of verification. When localized to quantitative properties of programs, **calf** evinces the notion of a *cost signature* [Acar and Blelloch 2019], the cost-aware counterpart to the functional specification of a data structure. In **calf**, we may specify the quantitative properties of a data structure by using *cost-aware* dependent functions $(a : A) \xrightarrow{c} B$, an application of the hasCost refinement from Section 1.4:

$$(a : A) \xrightarrow{c} B = \Sigma f : (\Pi a : A.\, B(a)).\, \Pi a : A.\, \mathsf{hasCost}(B(a), f(a), c(a))$$

Thus an element of $(a : A) \xrightarrow{c} B$ is a function f along with a proof that it satisfies the cost specification c on all instances.

To see this connective in action, consider clients Alice and Bob who both require a data structure to manipulate graphs. Alice may request a structure satisfying the left signature in Fig. 1, indicating that they would like edge insertion and membership to both be logarithmic in the number of vertices n. On the other hand, Bob's algorithm needs constant time edge membership, but is not so sensitive to changes to the graph. This requirement is captured by the right signature in Fig. 1.

Fortunately, both programmers can be supplied with suitable implementations: edge sets for Alice and adjacency matrices for Bob. Although somewhat artificial, this example shows that **calf** is able to formalize the notion of a cost signature as used by Acar and Blelloch [2019], paving the way to verified, cost-aware development of large-scale programs.

1.6 Analyzing the Cost of General Recursive Functions

Most efficient algorithms are not defined by structural induction on the input — their efficiency is the result of exploiting the structure of the data in clever, nonobvious ways that nevertheless terminate. It is not surprising that this often cannot be surmised by syntactic means and requires proof. Hence a type theoretic framework for cost analysis must provide a story for encoding general recursive algorithms such that the resulting analysis reflects the expected complexity (and not, for instance, the complexity of the termination proof).

A well-known and versatile solution to the encoding of general recursive functions in total type theory is the celebrated Bove–Capretta method [Bove and Capretta 2005]. Any general recursive program gives rise to an *accessibility predicate* that tracks the pattern of recursive calls; this accessibility predicate can be glued onto the original program as a termination metric, and the final (total) function is defined by proving that every input is accessible.

Cost recurrences provide an alternative to accessibility predicates. The idea is to parameterize a given program in a *clock*, induced by the cost recurrence, which can then serve as a termination

metric that frees the program to make whatever recursive calls are required. This strategy is attractive in the quantitative setting precisely because cost analysis computes the desired instantiation of the clock with no additional effort. In contrast the same method in a framework for pure behavioral properties becomes a technical device for definition that does not provide further insight into the defined program. As observed in Niu and Harper [2020], the cost-aware setting evinces a synergetic relationship between cost analysis itself and programming with general recursion that is further amplified in **calf**: cost structure enables one to effectively encode general recursion, and general recursion gives rise to programs with interesting cost structure.

Relationship to the normal form theorem. One of the most well-known results of computability theory Kleene [1943] is that any partial computable function of type $\mathbb{N} \to \mathbb{N}$ may be defined using one minimization operation; in other words, one "while loop" is sufficient to compute any partial function. We observe that the encoding of general recursive programs in **calf** shares a similar flavor in the sense that the call-graph of an encoded algorithm may be seen as counting down a single outer "for loop" whose bound is determined by the cost bound of the algorithm.

1.7 Cost Models and Adequacy

Informed by the actual practice of algorithm analysis, we do not associate a particular cost semantics to **calf** itself but instead promote the use of **calf** as a cost-aware metalanguage for expressing algorithm-specific/non-uniform cost models. Some authors (such as Danielsson) refer to this mode of reasoning as "semiformal" because there is no precise relationship to the traditional formulation of cost given by an operational semantics, which one may think of as a language-level/uniform cost model. As Danielsson [2008] points out, the connection between the uniform and non-uniform models can be made formal if one inserts "steps" at the right places, which is tantamount to programming in the *image* of a cost-preserving embedding of an object language equipped with an operational semantics. In Section 7 we conjecture that an adequacy result of this form may be proved for **calf** in the style of Kavvos et al. [2019]; Paviotti et al. [2015], constituting an internal version of the classic Plotkin-type adequacy theorem [Plotkin 1977] for **calf**.

From our point of view it is helpful to identify the uniform and non-uniform models as meaningful in their own contexts. For the purposes of algorithm analysis it is clearly preferable to work inside a framework that allows for different cost models for different classes of problems; cost models in this sense cannot be detected at the level of operational semantics — how would one delineate a comparison or edge insertion operation? On the other hand, it is ill-formed to speak of adequacy results such as Kavvos et al. [2019] with respect to a non-uniform cost model. Although in this paper we focus on the algorithms analysis perspective, we note that **calf** supports both perspectives because a uniform cost model is just an instance of a non-uniform cost model, and as we discussed above, the connection between the two can be made precise through an adequacy theorem.

1.8 Related Work

1.8.1 Recurrence Extraction Through CBPV. The CBPV decomposition of cost structure in **calf** is directly inspired by recent work on recurrence extraction [Kavvos et al. 2019] for functional programs. In that setting a source language such as CBV PCF is interpreted via a cost-preserving translation into CBPV, from which a *syntactic* recurrence relation is extracted; the syntactic recurrence is then translated into a semantic recurrence in a domain appropriate for mathematical manipulation used in algorithm analysis. The focus of this work is the formalization of the extract-and-solve paradigm used informally in algorithm analysis and the modularity with respect to the source language afforded by the CBPV decomposition.

Because the extraction process is stratified over different languages, the *bounding theorem* — the fact that a source program satisfies a syntactic recurrence relation — is an external fact. In contrast **calf** collapses the distinction between syntactic and semantic recurrence and is able to express the source program and the cost recurrence in the same language. Moreover, **calf** furnishes a rich specification language that allows us to prove internally that a program is bounded by a given cost.

Another difference is the presence of general recursion in the work on recurrence extraction. Because we propose **calf** as a logical framework for internal reasoning, inclusion of unrestricted fixed-points is a nonstarter. This does not, however, prevent us from analyzing the cost of general recursive programs: as discussed in Section 1.6, knowing the cost bound of an algorithm allows us to define it by recursion on cost in a total setting. Of course, cost bounds do not provide termination metrics for non-terminating programs; we expect that non-terminating programs can also be handled by means of a monad for partiality as in the work of Capretta [2005].

1.8.2 Effects in Dependent Type Theory. The key ingredient that endows **calf** with enough structure to serve as a logic for internal reasoning is the integration of dependent types in an effectful language. We essentially extend the universe-free fragment of the **∂cbpv** calculus of Pédrot and Tabareau [2019] by axioms for the extensional modality. The weaning translation of **∂cbpv** is the closest counterpart to the model we use to prove the soundness of **calf**. To define the weaning translation, Pédrot and Tabareau [2019] introduce the concept of the self-algebraic proto-monad, which provides the structure needed to model computation universes. Because we do not axiomatize universes, computation types in **calf** are interpreted as algebras over a strong monad as in the usual Eilenberg–Moore models of CBPV. To include universes, we expect that the **∂cbpv** approach can be further adapted to **calf** without significant modification.

1.8.3 Transparent vs. *Abstract Axiomatization of Cost Structure.* Semantically, free computations $\mathsf{F}(A)$ can be modeled as free algebras over the monad $\mathbb{C} \times -$. We are however careful to not commit to this fact *internally*; by keeping the type $\mathsf{F}(A)$ abstract, **calf** ensures that programs cannot drop costs or branch based on the cost component of their input.

As an example, a language that does not satisfy this noninterference property is the language of syntactic recurrences "PCF with costs" employed by Kavvos et al. [2019]. Indeed, by interpreting $\mathsf{F}(A)$ as $\mathbb{C} \times A$, the language of syntactic recurrences is made transparent enough that programs can spuriously use the cost of an input to choose an output. However, this is not an issue in that setting because of the stratification of the source language (of programs) and target language (of recurrences): such an exotic program lies outside the image of the interpretation.

1.8.4 Intensionality in Logic and Type Theory.

Intensional constructs in computational type theory. Cost structure in **calf** aims to capture an intensional aspect of programs, historically a difficult phenomenon to study type theoretically. Researchers in the **Nuprl** tradition have made a number of forays into intensionality beginning with the **PL/CV3** language, which included an operator isap that distinguished function applications from other terms [Constable and Zlatin 1984]. Constable and Crary [2002] later on introduce a version of type theory equipped with a more restricted form of intensionality by internalizing parts of the operational semantics, which can be construed as a form of reflective deep embedding.

Necessity modalities for intensionality. In the tradition of structural proof theory and modal type theory, the *necessity* modality $\Box A$ has been argued to capture the formal aspects of staged computation [Davies and Pfenning 1999]. From this perspective, $\Box A$ is the type of *codes* for terms of type A. A detailed investigation of this folklore was carried out by Kavvos [2017b], introducing the intensional PCF (iPCF) programming language with an intensional fixed-point operator whose

type is the Gödel-Löb axiom. Unfortunately, Kavvos's investigation revealed that truly intensional operations such as isap must be limited to syntactically closed terms; such a side condition casts doubt on the type theoretic nature of intensional operations. In the context of modal type theory, Pfenning [2001] investigates intensionality through a judgmental distinction between intensional expressions, extensional terms, and irrelevant proofs. However, the internalization of this new judgmental structure as modal operators is not fully worked out.

A common theme in prior work that aims to capture intensionality within type theory is that equations are *removed* underneath certain constructors, consequently refuting most congruence rules and obstructing presentations by generators and equations. Although tenable for simple theories, this approach greatly complicates the integration of type dependency, where congruence rules play a very important role in usability. In the design of **calf** we take the complementary perspective of *conditional extensionality*, where equations expressing extensional/behavioral properties are *added* in certain contexts. By modeling intension/extension as another phase distinction, we give an elegant mathematical account of the intensional content of programs without sacrificing extensionality principles or speaking of "equalities" that do not always hold.

1.8.5 *Type Systems for Cost Analysis.*

Amortized cost analysis. Many current type-theoretic approaches to cost analysis rely on the notion of *linearity*/non-duplicability of resources. A prototypical example is Hofmann's type system for programming in bounded space in which heap resources are abstracted into a type $\diamond$ that is required to construct heap-allocated data structures. This idea essentially started the line of work in automated amortized resource analysis (AARA) that includes automatic heap-space bounds [Hofmann and Jost 2003], analysis of higher-order programs [Jost et al. 2010], and a resource-aware version of OCaml (RaML) [Hoffmann et al. 2012]. More recently, the combination of AARA and temporal modalities has also been used in the setting of session types to analyze both sequential and parallel cost of message-passing programs [Das et al. 2018a,b; Das and Pfenning 2020].

In these type systems a derivation may be viewed as a *stateful* transformation of the context (*e.g.* consumable resource) into a computation that satisfies a cost bound, constituting a type-theoretic formulation of *amortized analysis* [Tarjan 1985]. Consequently, a linear/affine treatment of resources is critical for ensuring the soundness of quantitative reasoning, which states that the derived cost bound suffices for the actual cost as given by a standard cost dynamics.

As we discuss in Section 4.2, it is straightforward to formalize textbook formulations of amortized analysis in **calf** (we chose to demonstrate a particular formulation known as the *physicist's method*). However, it is not immediately clear how one may take better advantage of the existing type-based approaches to amortized analysis in **calf**.

Cost analysis of lazy programs. A common phenomenon of functional programming we do not consider in **calf** is lazy evaluation. A well-known type-based method for amortized analysis that also handles lazy evaluation is introduced through the Thunk library of Danielsson [2008], a lightweight semiformal approach to cost analysis based around a particular indexed monad *Thunk*. The rough idea is that an element of $\mathit{Thunk}\ n\ a$ is a term of type a that evaluates to weak head normal form in at most n steps.[2] One may give precise cost analyses of lazy programs in Thunk by encoding a version of Okasaki's banker's method Okasaki [1998] using *Thunk* types.

Similar to **calf**, the library allows users to freely place cost increments $\mathit{inc} : \mathit{Thunk}\ n\ a \to \mathit{Thunk}\ (1 + n)\ a$ to express different cost models. Because the *Thunk* monad is exposed to the user as an abstract interface, it is necessary to include an operation $\mathit{force} : \mathit{Thunk}\ n\ a \to a$ that

[2]Here we assume that a is a type that is not of the form $\mathit{Thunk}\ m\ b$ for some m and b; in general one has to consider the indices of all the nested *Thunk*'s.

facilitates the interaction of cost annotated terms and ordinary code. Clearly, *force* should not occur inside programs that are being analyzed, so in addition to ensuring that cost increments are placed correctly, one has to make sure that "running code" does not contain *force*. In contrast, although one also has to ensure the correct placement of step in **calf**, by design there is no operation analogous to *force* in **calf** that discards the cost component of computations.

Unlike **calf**, the framework of Danielsson [2008] does not handle verification of functional properties. As mentioned in *op. cit.*, it is difficult to work with types indexed in the *Thunk* monad, and it is unclear whether one can express complex behavioral properties of cost annotated terms.

Non-amortized cost analysis. Type theoretic formulations of cost analysis do not have to be based on amortization: Crary and Weirich [2000] develop a type system for resource bound certification by means of a *virtual clock*. Function types are refined with a starting and ending time, so that a function of type $(A, 5) \to (B, 0)$ is an ordinary function $A \to B$ with the property that it is to be applied when the clock is five and completes when the clock is zero. Clock polymorphism relaxes the limit on the starting time by allowing one to form the type $\forall n.\, (A, n+5) \to (B, n)$. Variable cost bounds are definable via a limited form of dependency using inductive kinds, which unfortunately imposes a somewhat stilted programming style.

More recently, Wang et al. [2017] introduced TiML, a language loosely based on Standard ML that provides internal cost specifications in the form of a timed function type. TiML supports indexed data types whose indices furnish a notion of size measure, leading to a more natural treatment of variable cost bounds compared to Crary and Weirich [2000]. The TiML type system generates verification conditions that are further refined by a recurrence solver using heuristics such as the Master Theorem [Cormen et al. 2009].

Type systems presented in Crary and Weirich [2000]; Wang et al. [2017] represent practical compromises in the sense that they are primarily designed for expressing cost information and only secondarily support limited forms of behavioral specification. In contrast **calf** is a full-spectrum dependent type theory designed for *both* quantitative and behavioral verification.

Frameworks for cost analysis. **calf** is a *framework* for cost analysis in the sense that it provides the language for speaking about the cost structure of programs but does not prescribe a particular method for cost analysis. Recently, Rajani et al. [2021] advance a similar thesis by developing a type theory, λ-amor, that unifies many extant type systems for cost analysis, in particular exhibiting λ-amor embeddings of both effect and coeffect-based systems for cost accounting. However, because λ-amor does not support dependent types, there is no satisfying account of the behavioral fragment.

In the context of Liquid Haskell, Handley et al. [2019] define a monadic library called RTick for reasoning about both quantitative and correctness properties by taking advantage of Liquid Haskell's refinement type system. They substantiate the library with a rich repository of examples, including sorting algorithms, programs optimizations, and relational cost analysis. However, because cost is represented transparently via the cost monad in the library, one may define exotic programs in the sense of Section 1.8.3 that use the cost of an input to choose the behavior of an output.

Finally, we mention the work of Niu and Harper [2020] on a cost-aware computational type theory **CATT** in the **Nuprl** tradition. Unlike the type theory of Constable and Crary [2002], **CATT** only internalizes cost structure, which leads to a framework that is more directly applicable to cost analysis. In particular, Niu and Harper [2020] introduce a connective "funtime" that internalizes cost specification on functions types and prove a novel refinement rule for funtime by appealing to the specified cost bound, constituting an induction principle based on cost structure. Our observation that cost analysis may be used to encode general recursion in **calf** is inspired by the work on **CATT**, as is the idea of using a cost-aware dependent function type to specify cost signatures. Niu

and Harper [2020] do not develop a formal proof theory for **CATT**, a fact that appears to pose significant challenges for its mechanization.

1.8.6 Separation Logic. An alternative perspective, exemplified by the work of Atkey [2010] on amortized resource analysis in separation logic, is to treat cost as an ownable resource. Program logics in this tradition primarily focus on the verification of imperative programs. Atkey's formulation essentially transposes the types-with-potential concept of Hofmann and Jost [2003] into the imperative setting, allowing one to prove resource bounds on heap-based data structures.

More recently, Mével et al. [2019] employed similar ideas to develop a resource-aware extension to the Iris program logic [Jung et al. 2018, 2015]. The interesting twist in this work is the use of time *receipts*, which are dual to the more common time *credits*. Time receipts witness that a computation takes at least a certain amount of resources, thereby establishing a lower bound on the cost of programs. This can be used to prove that catastrophic events do not happen until a long time has passed. An application of the framework is the verification of an asymptotically tight upper bound on union-find, a mathematically involved and complex proof.

Iris is a very powerful tool whose scope goes far beyond cost analysis; the theoretical overhead of Iris when applied *specifically* to quantitative analysis of functional programs is consequently somewhat high in contrast to the basic rules of **calf** which can be written down in half a page. Furthermore, the intended semantics of **calf** can be interpreted somewhat simple-mindedly in *any* topos equipped with a subterminal sheaf representing the partition between extension and intension. In this respect, **calf** offers a fundamentally different perspective on cost analysis based on the *synthetic* integration of cost specification into a full-spectrum dependent type theory rather than the definition of a resource-sensitive program logic over an existing language.

1.8.7 Isabelle/HOL. The proof assistant Isabelle/HOL represents another hot spot for complexity verification. In this setting the *Archive of Formal Proofs* contains a number of case studies on complexity verification, including quicksort [Eberl 2017b], medians of medians [Eberl 2017a], and the formalization of the Akra-Bazzi theorem [Eberl 2015], just to name a few. In more recent work Nipkow et al. [2021] give a systematic study of the functional correctness and complexity verification of a variety of algorithms and techniques including sorting, search trees, amortized analysis, dynamic programming, *etc.*

In these works cost is often instrumented through the writer monad $\mathbb{N} \times -$ or just treated informally. In contrast, **calf** allows the user to define formal relations between programs and recurrences, and the careful instrumentation of cost structure as a computational effect induces a noninterference property not found in the Isabelle/HOL setting. The Isabelle/HOL approach to cost analysis uses existing tools in the framework to *encode* the notion of cost, while **calf** is a framework in which one can use type-theoretic principles to reason about cost/quantitative properties of programs in a first class way without sacrificing the connection to the uninstrumented programs.

2 COST-AWARE LOGICAL FRAMEWORK

We define **calf** as an extension to the **∂cbpv** calculus of Pédrot and Tabareau [2019]. As discussed in Section 1, the design of **calf** rests on three main pillars. First, the fine-grained type structure of CBPV gives a compositional account of cost at higher types. Secondly, in the dependent setting **∂cbpv** provides a smooth integration of effects and type dependency, which allows us to define cost-aware programs and prove theorems about them in a single language. Lastly (as in Section 1.3), the extensional phase $¶_{\mathsf{E}}$ generates a pair of complementary open and closed modalities $\bigcirc, \bullet$ in the sense of Rijke et al. [2020]; Schultz and Spivak [2019] that govern the interaction between *intension* and *extension*. In the following, we introduce **calf** at an informal level through simple

examples that illustrate the cost effect step, internal cost bounds as equations, and the interplay of the *open/extensional modality* $\bigcirc$ and the *closed/intensional modality* $\bullet$.

2.1 A Refresher on CBPV: the Identity Function Two Ways

We give a quick introduction to CBPV through the simplest possible example: the identity function (on natural numbers). Recall that the type structure of CBPV is centered around the polarization of values and computations. For our example, consider the following selection of types and terms:[3]

Values	**Computations**
$A, B \coloneqq \mathsf{U}(X), \mathsf{nat}$	$X, Y \coloneqq \mathsf{F}(A), A \to X$
$a, b \coloneqq \mathsf{thunk}(e), \mathsf{zero}, \mathsf{suc}(a)$	$e, f \coloneqq \mathsf{ret}(a), \mathsf{bind}(e; f), \mathsf{force}(u), \mathsf{rec}(a)\{e_1; e_2\}, \lambda a.\, e, \mathsf{ap}(f; v)$

The pair of type constructors F and U bridges the dichotomy between value and computation types: F turns a value $a : A$ into the computation $\mathsf{ret}(a) : \mathsf{F}(A)$, and U reifies a computation $e : X$ into a value $\mathsf{thunk}(e) : \mathsf{U}(X)$. Observe that functions are *computations* in CBPV, a phenomenon that may be explained by examining the operational behavior of functions in the CK-machine model of CBPV [Levy 2006]. The fine-grained type structure of CBPV evinces embeddings of both CBV and CBN. For instance, one may recover the CBV function space $\mathsf{nat} \to_{\mathsf{cbv}} \mathsf{nat}$ as the CBPV type $\mathsf{U}(\mathsf{nat} \to \mathsf{F}(\mathsf{nat}))$. We refer the reader to Levy [2004] for a more thorough introduction.

For the purposes of our example, we only consider the value type nat. As usual, zero and $\mathsf{suc}(n)$ are values of nat. The recursor is assigned the type $\mathsf{rec} : \{X\}\ \mathsf{nat} \to X \to (\mathsf{nat} \to \mathsf{U}(X) \to X) \to X$. Note that the recursive call is reified as a value $\mathsf{U}(X)$ because variables range over values in CBPV. If we restrict attention to natural numbers, there are two evident ways to compute the identity: one program returns the argument immediately, and the other reconstructs the argument by recursion. In CBPV, they are rendered as the following programs:

$$id_{\mathsf{easy}} : \mathsf{nat} \to \mathsf{F}(\mathsf{nat}) \qquad id_{\mathsf{hard}} : \mathsf{nat} \to \mathsf{F}(\mathsf{nat})$$
$$id_{\mathsf{easy}} = \lambda x.\, \mathsf{ret}(x) \qquad id_{\mathsf{hard}} = \lambda x.\, \mathsf{rec}(x)\{\mathsf{ret}(\mathsf{zero}); \lambda x', u.\, \mathsf{bind}(\mathsf{force}(u); \lambda y.\, \mathsf{ret}(\mathsf{suc}(y)))\}$$

Note that in id_{hard} we have to force the reified recursive computation $u : \mathsf{U}(\mathsf{F}(\mathsf{nat}))$ to obtain a computation $\mathsf{F}(\mathsf{nat})$, thence sequencing it and tacking on an additional successor.

2.2 Cost Monoid: Cost Structure of Programs

Cost-aware programs carry quantitative information through elements of the cost monoid $\mathbb{C}$. Because different algorithms and cost models require different notions of cost, we parameterize **calf** by an arbitrary *cancellative monoid* $(\mathbb{C}, +, 0)$; here cancellative means that the operation $+$ is injective, a property that is needed to establish metatheoretic results in Section 5. Further structure on $\mathbb{C}$ can be negotiated depending on one's preference for generality. For the purposes of analyzing (upper) bounds of algorithms, it is reasonable to additionally require the structure of an *ordered monoid* $(\mathbb{C}, +, 0, \leq)$ in which the monoid multiplication is compatible with a preorder $\leq$.

2.3 Cost as an Effect in calf

We formulate cost in **calf** as a primitive effect by adding a new form of computation $\mathsf{step}^c_X(e)$ that is parameterized by a computation type X and an element of the cost monoid c. The meaning of $\mathsf{step}^c_X(e)$ is to effect c units of cost and continue as e; consequently, we require that step is coherent with the monoid structure on $\mathbb{C}$:

$$\mathsf{step}^0_X(e) = e \qquad \mathsf{step}^c_X(\mathsf{step}^d_X(e)) = \mathsf{step}^{c+d}_X(e)$$

[3] **calf** also includes additional types such as dependent products and dependent sums.

In addition, we require a slew of equations governing the interaction of step with other computations. For instance, step satisfies the following laws:

$$\mathsf{bind}_{\mathsf{step}} : \{e : \mathsf{F}(A), f : A \to X\}\ \mathsf{bind}(\mathsf{step}^c_{\mathsf{F}(A)}(e); f) = \mathsf{step}^c_X(\mathsf{bind}(e; f))$$
$$\mathsf{lam}_{\mathsf{step}} : \mathsf{step}^c_{A\to X}(\lambda x.\, e) = \lambda x.\, \mathsf{step}^c_X(e)$$

The first equation states that step inside a sequence of computations can be commuted outside and executed first; the second equation states that step commutes with abstraction.

Meaning of step *in the Eilenberg–Moore model of* ***calf****.* Each computation type X of **calf** is interpreted as an algebra $(|X|, \alpha)$ over $\mathbb{C} \times -$. Thus step_X is interpreted by the structure map α, and all of the equations associated with step_X hold as a consequence of the algebra laws.

Cost of identity. For the identity example, let us suppose that $\mathbb{C}$ is the additive monoid on $\mathbb{N}$ under the usual ordering. Consider the two identity programs from the previous section. Suppose that we wanted to charge unit cost for each recursive call in the program. In **calf**, we can achieve this by instrumenting the program with step at the appropriate place:

$$id_{\mathsf{hard}} = \lambda x.\, \mathsf{rec}(x)\{\mathsf{ret}(\mathsf{zero}); \lambda x', u.\, \mathsf{step}^1_{\mathsf{F}(\mathsf{nat})}(\mathsf{bind}(\mathsf{force}(u); \lambda y.\, \mathsf{ret}(\mathsf{suc}(y))))\}$$

We do nothing for id_{easy} because there is no recursion involved.

2.4 Cost Refinements in calf

Recall the predicate hasCost from Section 1, $\mathsf{hasCost}(A, e, c) \coloneqq \Sigma a : A.\, e =_{\mathsf{F}(A)} \mathsf{step}^c(\mathsf{ret}(a))$, which states that the computation $e : \mathsf{F}(A)$ incurs c units of cost. Given our instrumented identities, we can prove the following quantitative refinements for id_{easy} and id_{hard}:

Theorem 2.1 (`Examples.Id.Easy.id≤id/cost`). *We have that $id_{\mathsf{easy}}(x)$ has cost 0 for all x : nat.*

Proof. We take the input as the witness value and apply the coherence rule of step to obtain $id_{\mathsf{easy}}(x) = \mathsf{ret}(x) = \mathsf{step}^0_X(\mathsf{ret}(x))$. □

Theorem 2.2 (`Examples.Id.Hard.id≤id/cost/closed`). *We have that $id_{\mathsf{hard}}(x)$ has cost $\iota(x)$ for all x : nat, where ι is the obvious monoid isomorphism* $\mathsf{nat} \cong \mathbb{N}$.

Proof. We proceed by induction on x. In the inductive case, we use the equations governing step explained in Section 2.3 and the inversion principles for U and F. □

The study of quantitative properties *qua* equations evinces the essential advantage of verification in **calf**: proof of quantitative properties is reduced to ordinary equational reasoning.

2.5 Reasoning About Extensional Properties Using $\P_{\mathsf{E}}$

In general, equations between cost-aware programs of **calf** are in some sense rare, exactly because the cost effect obstructs equations between extensionally equivalent computations. To account for extensional equivalence and other behavioral properties, we study programs in the fragment of **calf** under the extensional phase $\P_{\mathsf{E}}$.

The extensional fragment of ***calf****.* As discussed in Section 1.3, the (proof-irrelevant) proposition $\P_{\mathsf{E}}$ renders the extensional modality as the function space $\bigcirc A \coloneqq \P_{\mathsf{E}} \to A$, which naturally generalizes to a dependent modality $\bigcirc_{u:\P_{\mathsf{E}}}(A(u)) \coloneqq (u : \P_{\mathsf{E}}) \to A(u)$. The force of this modality is effected by the following axiom in **calf**, which makes step silent in the presence of $\P_{\mathsf{E}}$:

$$\mathsf{step}/\P_{\mathsf{E}} : \bigcirc(\mathsf{step}^c_X(e) = e)$$

data $\bullet A$ **where**	$\bigcirc(A^{\Downarrow c} \to B^{\Downarrow d}) \cong \bigcirc A^{\Downarrow c} \to \bigcirc B^{\Downarrow d}$
$\eta_\bullet : A \to \bullet A$	$= \bigcirc(\Sigma e : A.\ \bullet\mathsf{hasCost}(A, e, c)) \to \bigcirc(\Sigma f : B.\ \bullet\mathsf{hasCost}(B, f, d))$
$* : \P_\mathsf{E} \to \bullet A$	$\cong \Sigma e : \bigcirc A.\ \bigcirc_{u:\P_\mathsf{E}} \bullet\mathsf{hasCost}(A, e(u), c) \to \Sigma f : \bigcirc B.\ \bigcirc_{u:\P_\mathsf{E}} \bullet\mathsf{hasCost}(B, f(u), d)$
$\_ : \Pi a : A.\, \Pi u : \P_\mathsf{E}.$	$\cong (\Sigma e : \bigcirc A.\ 1) \to (\Sigma f : \bigcirc B.\ 1)$
$\eta_\bullet(a) = *(u)$	$\cong \bigcirc A \to \bigcirc B \cong \bigcirc(A \to B)$

Fig. 2. Left: closed modality as a quotient inductive type; right: extracting the extensional content of cost-aware functions, where $\bigcirc_{u:\P_\mathsf{E}} A(u) \coloneqq (u : \P_\mathsf{E}) \to A(u)$ is the dependent version of the extensional modality.

Thus the extensional modality $\bigcirc$ governs *behavioral* specifications in the sense that any type in the image of $\bigcirc$ is oblivious to computation steps. One such behavioral specification is the *extensional equality* between programs, rendered in **calf** as the type $\bigcirc(e_1 = e_2)$. In the case of the two identity programs, we can take id_{easy} as the specification and prove that id_{hard} obeys it:

Theorem 2.3 (`Examples.Id.easy≡hard`). *We have the modal equation* $\bigcirc(id_{\mathsf{hard}} = id_{\mathsf{easy}})$.

Proof. By function extensionality, it suffices to show $id_{\mathsf{hard}}(x) = id_{\mathsf{easy}}(x)$ for all x : nat and $u : \P_\mathsf{E}$. This follows by induction on x, using the equation $\mathsf{step}/\P_\mathsf{E}(u)$ in the inductive case. □

2.6 Closed/Intensional Modality

Complementary to the open/extensional modality $\bigcirc$ is the *closed/intensional modality* $\bullet$ that governs intensional/quantitative properties. One may think of applying the intensional modality as sealing away the extensional part so it cannot be observed and leaving only the intensional part of a program. Consequently a type in the image of the intensional modality $\bullet$ is trivial under the extensional modality $\bigcirc$, *i.e.* $\bigcirc\bullet A \cong 1$ for any type A. In the Eilenberg–Moore model of **calf**, we exploit this property to enforce the step erasure rule $\mathsf{step}/\P_\mathsf{E}$: by interpreting computation types as algebras for the writer monad $\bullet\mathbb{C} \times -$, the cost structure of programs is obliterated whenever the extensional phase $\P_\mathsf{E}$ is present in the context. In Fig. 2 we define the intensional modality as a quotient inductive type [Altenkirch and Kaposi 2016b; Fiore et al. 2021]. In categorical language the intensional modality is the pushout $A \sqcup_{A \times \P_\mathsf{E}} \P_\mathsf{E}$ of the projection maps of $A \times \P_\mathsf{E}$.

Program extraction. The intensional modality allows one to organize quantitative information in a way that facilitates extraction of ordinary programs from cost-aware programs. For instance, consider the type of functions between cost-aware computations $A^{\Downarrow c} \to B^{\Downarrow d}$ where $A^{\Downarrow c} \coloneqq \Sigma e : A.\ \bullet\mathsf{hasCost}(A, e, c)$ is the type of computations of A that incur c steps. Note that the type hasCost is guarded by the intensional modality. Fig. 2 shows that one may extract the underlying function by applying the *extensional modality* $\bigcirc$ and using the fact that $\bigcirc$ is lex and commutes with exponentials.

2.7 Noninterference

In Section 1 we claim that the extensional part of **calf** programs cannot analyze the intensional part of their input. This is substantiated by the following theorem:

Theorem 2.4 (`Calf.Noninterference.oblivious`). *Given any function* $f : \mathsf{F}(A) \to \bigcirc B$, *we have that* $f(\mathsf{step}^c(e)) = f(e)$ *for any* $c : \mathbb{C}$ *and* $e : \mathsf{F}(A)$.

Moreover, when the input of a **calf** program is fully intensional, we may obtain a stronger noninterference property by observing the interaction of the extensional and intensional modalities (also exploited in the program extraction example in Section 2.6):

Theorem 2.5 (`Calf.Noninterference.constant`). *Any function* $f : \bullet A \to \bigcirc B$ *is constant.*

In other words one cannot construct a map into the extensional fragment that branches based on purely intensional information, a fact that enables a type-directed method to systematically eliminate intensional structure from programs of a certain shape. For instance, one may perform the following *program optimization*:

Theorem 2.6 (`Calf.Noninterference.optimization`). *Any map* $f : (\Sigma c : C.\, \bullet A(c)) \to \bigcirc B$ *admits an optimization* $f' : C \to \bigcirc B$ *such that* $f(c, a) = f'(c)$ *for all* $c : C$ *and* $a : \bullet A(c)$.

Proof. We need to construct a map $f' : C \to \bigcirc B$. Suppose that $c : C$. By Theorem 2.5, there is a constant map $\lambda\_.\, b : \bullet A(c) \to \bigcirc B$ such that $\lambda a.\, f(c, a) = \lambda\_.\, b$, which provides the required program b of type $\bigcirc B$. By definition, we have $f(c, a) = f'(c)$ for all $c : C$ and $a : \bullet A(c)$. □

Recalling the type of bounded computations $A^{\Downarrow c}$ from Section 2.6, we may apply Theorem 2.6 to a program of type $A^{\Downarrow c} \to \bigcirc B$ to obtain an optimized program of type $A \to \bigcirc B$ that dispenses with the proof of the cost bound.

2.8 Presentation of calf in a Logical Framework

Following recent work [Gratzer and Sterling 2020; Sterling and Harper 2021; Uemura 2019] promoting the study of type theories *qua* mathematical objects in structured categories, we present **calf** as a signature in a *logical framework* using the internal language of locally cartesian closed categories (lcccs). As observed by Uemura [2019], one can specify a type theory as a list of constants in a version of extensional dependent type theory. The resulting signature *presents* the free lccc over the defined constants, which we then take as the *definition* of the type theory. As we show in Section 5, this view of type theories as certain initial objects allows one to easily define models of **calf**.

Concretely, we work in a logical framework with a universe of judgments **Jdg** closed under dependent product, dependent sum, and extensional equality. An object theory (*e.g.* **calf**) is specified as follows:

(1) Judgments are declared as constants ending in **Jdg**.
(2) Binding and scope is handled by the framework-level dependent product $(x : X) \to Y(x)$.
(3) Equations between object-level terms are specified by constants ending in the framework-level equality type $x_1 =_X x_2$.

*Presentation of **calf** in the logical framework.* We present **calf** in Fig. 3. For brevity, we do not explicitly mention all types and computations here, the majority of which remain unchanged from **∂cbpv**. Note that we define computations as $\mathsf{tm}^{\ominus}(X) \coloneqq \mathsf{tm}^{+}(\mathsf{U}(X))$, leading to a less bureaucratic version of CBPV in which thunk and force are identities. The **calf** equality type eq comes equipped with a reflection rule ref that renders inhabitation of eq equi-derivable with judgmental equality. Thus we abuse notation slightly and also write $e =_{\mathsf{tm}^{\ominus}(X)} f$ for the type $\mathsf{eq}_{\mathsf{U}(X)}(e, f)$.

*Presentation of **calf** with contexts.* One may also present **calf** in a more traditional style that defines mutually inductively three basic judgments (ignoring equality judgments) governing contexts $\boxed{\Gamma\ \mathsf{ctx}}$, types $\boxed{\Gamma \vdash A\ \mathsf{tp}^{\pm}}$, and terms $\boxed{\Gamma \vdash e : \mathsf{tm}^{\pm}(A)}$. In particular, a well-formed context in such a presentation of **calf** would only contain structural assumptions of the form $a : \mathsf{tm}^{\pm}(A)$ or $u : \P_{\mathsf{E}}$. Note that although we have promoted the terminology of "modalities" when describing **calf**, the introduction and elimination rules of the extensional/intensional modalities behave like ordinary connectives of type theory and involve only standard type theoretic contexts with structural substitutions. For instance, the extensional/open modality may be defined as follows:

$$\bigcirc\text{-F}\ \frac{\Gamma \vdash A\ \mathsf{tp}^{+}}{\Gamma \vdash \bigcirc^{+} A\ \mathsf{tp}^{+}} \qquad \bigcirc\text{-I}\ \frac{\Gamma, u : \P_{\mathsf{E}} \vdash e : \mathsf{tm}^{+}(A)}{\Gamma \vdash \bigcirc_{\mathsf{in}}(e) : \mathsf{tm}^{+}(\bigcirc^{+} A)} \qquad \bigcirc\text{-E}\ \frac{\Gamma \vdash e : \mathsf{tm}^{+}(\bigcirc^{+} A) \qquad \Gamma \vdash u : \P_{\mathsf{E}}}{\Gamma \vdash \bigcirc_{\mathsf{out}}(e, u) : \mathsf{tm}^{+}(A)}$$

In the logical framework presentation of **calf** the extensional modality is more economically defined by the following constants (also displayed in Fig. 3):

$$\bigcirc^+ : \mathsf{tp}^+ \to \mathsf{tp}^+$$
$$\_ : \{A\}\ \mathsf{tm}^+(\bigcirc^+ A) \cong \bigcirc(\mathsf{tm}^+(A))$$

The adequacy of the logical framework presentation of **calf** with respect to the presentation with contexts is a result of recent work on defining dependent type theories in the doctrine of lcccs [Gratzer and Sterling 2020].

3 QUANTITATIVE REFINEMENT IN calf

In Section 2.8 we developed the skeletal structure of **calf** equipped the effect step for cost instrumentation. In this section, we define a quantitative refinement expressing the *upper bound* of a computation and present a collection of expected rules for the refinement relation. As mentioned in Section 1, this mode of quantitative reasoning manifests as *equations* between computations; we can make meaningful inferences about the cost of a computation e by equating it to another computation whose cost structure is readily available, *i.e.* $\mathsf{step}^c(\mathsf{ret}(a))$.

As a first attempt, we may conjecture that a computation $e : \mathsf{tm}^{\ominus}(\mathsf{F}(A))$ is bounded by $c : \mathbb{C}$ if $e =_{\mathsf{tm}^{\ominus}(\mathsf{F}(A))} \mathsf{step}^{c'}(\mathsf{ret}(a))$ for some $c' \leq c$ and $a : \mathsf{tm}^+(A)$. While this is a perfectly sensible definition, our investigations suggest it is more natural to replace ordinary inequality $\leq$ with the *extensional inequality* $\bigcirc(c' \leq c)$. Consequently the upper bound specification may be concisely expressed by refining the hasCost refinement from Section 1.4:

$$\mathsf{hasCost}(A, e, c) = \Sigma^{++} a : A.\ e =_{\mathsf{tm}^{\ominus}(\mathsf{F}(A))} \mathsf{step}^c(\mathsf{ret}(a))$$
$$\mathsf{isBounded}(A, e, c) = \Sigma^{++} c' : \mathsf{U}(\widehat{\mathbb{C}}).\ \bigcirc^+(\mathsf{U}(c' \mathrel{\widehat{\leq}} c)) \times \mathsf{hasCost}(A, e, c')$$

Here $\widehat{\mathbb{C}}$ and $\mathrel{\widehat{\leq}}$ internalizes the (judgmental) structure of the cost monoid $\mathbb{C}$ as **calf** types, *i.e.* we have that $\mathsf{tm}^{\ominus}(\widehat{\mathbb{C}}) \cong \mathbb{C}$. The use of the extensional inequality in the isBounded refinement reflects the intuition that "costs don't have cost". More importantly, this arrangement grants one access to the extensional fragment and the *extensional* properties therein when proving cost refinements, which is essential for analyses of algorithms that depend on behavioral invariants of data structures. In Section 5, we prove that "extensional cost bounds" $\bigcirc(c \leq c')$ are equivalent to ordinary cost bounds $c \leq c'$ for a large class of cost monoids in the intended model of **calf**.

3.1 Quantitative Refinement Rules

calf admits many expected principles for reasoning about the isBounded refinement. We present the exemplary rules used in the case studies in Section 4, summarized in inference rule style in Fig. 4. There are three syntax-directed refinements: the Return refinement bounds the return of a value by the neutral element $0 : \mathbb{C}$; the Step refinement states that step^c increases the bound on a computation by c; the Bind refinement combines the bounds on a sequence of computations. Lastly, the Relax refinement allows a cost bound to be replaced with a weaker bound.

3.2 Recursion

As mentioned in Section 1.6, Bove and Capretta's accessibility predicates provide a way to express general recursive definitions in type theory. Inspired by Niu and Harper [2020], we provide an alternative approach in **calf** that exploits the cost structure of programs: one can use the cost bound of a given algorithm to *safely define* the algorithm in question. Instead of accessibility predicates, we may parameterize every program by a *clock* that represents the amount of fuel available for recursion. We say that an instantiation of the clock is *safe* when it provides enough fuel for the

$$\begin{aligned}
\mathbb{C} &: \mathbf{Jdg} \\
0 &: \mathbb{C} \\
+ &: \mathbb{C} \to \mathbb{C} \to \mathbb{C} \\
\leq &: \mathbb{C} \to \mathbb{C} \to \mathbf{Jdg} \\
\mathsf{costMon} &: \mathsf{isCostMonoid}(\mathbb{C}, 0, +, \leq) \\
\mathsf{step} &: \{X : \mathsf{tp}^{\ominus}\}\ \mathbb{C} \to \mathsf{tm}^{\ominus}(X) \to \mathsf{tm}^{\ominus}(X) \\
\mathsf{step}_0 &: \{X, e\}\ \mathsf{step}^0(e) = e \\
\mathsf{step}_+ &: \{X, e, c_1, c_2\} \\
&\quad \mathsf{step}^{c_1}(\mathsf{step}^{c_2}(e)) = \mathsf{step}^{c_1+c_2}(e)
\end{aligned}$$

$$\begin{aligned}
\mathsf{tp}^+ &: \mathbf{Jdg} \\
\mathsf{tm}^+ &: \mathsf{tp}^+ \to \mathbf{Jdg} \\
\mathsf{U} &: \mathsf{tp}^{\ominus} \to \mathsf{tp}^+ \\
\mathsf{F} &: \mathsf{tp}^+ \to \mathsf{tp}^{\ominus} \\
\mathsf{tm}^{\ominus}(X) &\coloneqq \mathsf{tm}^+(\mathsf{U}(X)) \\
\mathsf{ret} &: (A : \mathsf{tp}^+, a : \mathsf{tm}^+(A)) \to \mathsf{tm}^{\ominus}(\mathsf{F}(A)) \\
\mathsf{bind} &: \{A : \mathsf{tp}^+, X : \mathsf{tp}^{\ominus}\}\ \mathsf{tm}^{\ominus}(\mathsf{F}(A)) \to \\
&\quad (\mathsf{tm}^+(A) \to \mathsf{tm}^{\ominus}(X)) \to \mathsf{tm}^{\ominus}(X)
\end{aligned}$$

$$\begin{aligned}
\P_{\mathsf{E}} &: \mathbf{Jdg} \\
\P_{\mathsf{E}}/\mathsf{uni} &: \{u, v : \P_{\mathsf{E}}\}\ u = v
\end{aligned}$$

$$\begin{aligned}
\bigcirc(\mathcal{J}) &\coloneqq \P_{\mathsf{E}} \to \mathcal{J} \\
\mathsf{step}/\P_{\mathsf{E}} &: \{X, e, c\}\ \bigcirc(\mathsf{step}^c(e) = e) \\
\bigcirc^+ &: \mathsf{tp}^+ \to \mathsf{tp}^+ \\
_ &: \{A\}\ \mathsf{tm}^+(\bigcirc^+ A) \cong \bigcirc(\mathsf{tm}^+(A))
\end{aligned}$$

$$\begin{aligned}
\bullet &: \mathsf{tp}^+ \to \mathsf{tp}^+ \\
\eta_\bullet &: \mathsf{tm}^+(A) \to \mathsf{tm}^+(\bullet A) \\
* &: \P_{\mathsf{E}} \to \mathsf{tm}^+(\bullet A) \\
_ &: \Pi a : \mathsf{tm}^+(A).\, \Pi u : \P_{\mathsf{E}}.\, \eta_\bullet(a) = *(u) \\
\mathsf{ind}_\bullet &: \{A\}\ (a : \mathsf{tm}^+(\bullet A)) \to (X : \mathsf{tm}^+(\bullet A) \to \mathsf{tp}^{\ominus}) \to \\
&\quad (x_0 : (a : \mathsf{tm}^+(A)) \to \mathsf{tm}^{\ominus}(X(\eta_\bullet(a)))) \to \\
&\quad (x_1 : (u : \P_{\mathsf{E}}) \to \mathsf{tm}^{\ominus}(X(*(u)))) \to \\
&\quad ((a : \mathsf{tm}^+(A)) \to (u : \P_{\mathsf{E}}) \to x_0(a) = x_1(u)) \to \\
&\quad \mathsf{tm}^{\ominus}(X(a))
\end{aligned}$$

$$\begin{aligned}
\Pi &: (A : \mathsf{tp}^+, X : \mathsf{tm}^+(A) \to \mathsf{tp}^{\ominus}) \to \mathsf{tp}^{\ominus} \\
(\mathsf{ap}, \mathsf{lam}) &: \{A, X\}\ \mathsf{tm}^{\ominus}(\Pi(A; X)) \cong (a : \mathsf{tm}^+(A)) \to \mathsf{tm}^{\ominus}(X(a)) \\
\Sigma^{++} &: (A : \mathsf{tp}^+, B : \mathsf{tm}^+(A) \to \mathsf{tp}^+) \to \mathsf{tp}^+ \\
(\mathsf{unpair}^{++}, \mathsf{pair}^{++}) &: \{A, B\}\ \mathsf{tm}^+(\Sigma^{++}(A; B)) \cong \Sigma(\mathsf{tm}^+(A))(\lambda a.\, \mathsf{tm}^+(B(a))) \\
\Sigma^{+-} &: (A : \mathsf{tp}^+, X : \mathsf{tm}^+(A) \to \mathsf{tp}^{\ominus}) \to \mathsf{tp}^{\ominus} \\
(\mathsf{unpair}^{+-}, \mathsf{pair}^{+-}) &: \{A, X\}\ \mathsf{tm}^{\ominus}(\Sigma^{+-}(A; X)) \cong \Sigma(\mathsf{tm}^+(A))(\lambda a.\, \mathsf{tm}^{\ominus}(X(a)))
\end{aligned}$$

$$\begin{aligned}
\mathsf{eq} &: (A : \mathsf{tp}^+) \to \mathsf{tm}^+(A) \to \mathsf{tm}^+(A) \to \mathsf{tp}^+ \\
\mathsf{self} &: \{A\}\ (a, b : \mathsf{tm}^+(A)) \to \\
&\quad a =_{\mathsf{tm}^+(A)} b \to \mathsf{tm}^+(\mathsf{eq}_A(a, b)) \\
\mathsf{ref} &: \{A\}\ (a, b : \mathsf{tm}^+(A)) \to \\
&\quad \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{eq}_A(a, b))) \to a =_{\mathsf{tm}^+(A)} b \\
\mathsf{uni} &: \{A, a, b\}\ (p, q : \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{eq}_A(a, b)))) \to \bigcirc(p = q)
\end{aligned}$$

$$\begin{aligned}
\mathsf{nat} &: \mathsf{tp}^+ \\
\mathsf{zero} &: \mathsf{tm}^+(\mathsf{nat}) \\
\mathsf{suc} &: \mathsf{tm}^+(\mathsf{nat}) \to \mathsf{tm}^+(\mathsf{nat}) \\
\mathsf{rec} &: (n : \mathsf{tm}^+(\mathsf{nat})) \to \\
&\quad (X : \mathsf{tm}^+(\mathsf{nat}) \to \mathsf{tp}^{\ominus}) \to \mathsf{tm}^{\ominus}(X(\mathsf{zero})) \to \\
&\quad ((n : \mathsf{tm}^+(\mathsf{nat})) \to \mathsf{tm}^{\ominus}(X(n)) \to \\
&\quad\quad \mathsf{tm}^{\ominus}(X(\mathsf{suc}(n)))) \to \mathsf{tm}^{\ominus}(X(n))
\end{aligned}$$

$$\begin{aligned}
\mathsf{lam}_{\mathsf{step}} &: \{A, X, f, c\}\ \mathsf{lam}(\mathsf{step}^c(f)) = \mathsf{step}^c(\mathsf{lam}(f)) \\
\mathsf{pair}_{\mathsf{step}} &: \{A, X, e_1, e_2, c\}\ \mathsf{step}^c((e_1, e_2)) = (e_1, \mathsf{step}^c(e_2)) \\
\mathsf{bind}_{\mathsf{step}} &: \{A, X, e, f, c\}\ \mathsf{bind}(\mathsf{step}^c(e); f) = \mathsf{step}^c(\mathsf{bind}(e; f))
\end{aligned}$$

Fig. 3. Equational presentation of **calf** as a signature $\Sigma_{\mathbf{calf}}$ in the logical framework. Here the type isCostMonoid encodes all the structure of a cost monoid and Σ denotes the framework-level dependent sum. We write $(\alpha, \beta) : A \cong B$ when α and β are the forward map and backward map of an isomorphism $A \cong B$.

$$\textsc{Return}\ (\texttt{Calf.Types.Bounded.bound/ret})\quad \frac{}{\mathsf{isBounded}\,(A; \mathsf{ret}(a); 0)}$$

$$\textsc{Step}\ (\texttt{Calf.Types.Bounded.bound/step})\quad \frac{\mathsf{isBounded}\,(A; e; d)}{\mathsf{isBounded}\,(A; \mathsf{step}^c(e); c + d)}$$

$$\textsc{Bind}\ (\texttt{Calf.Types.Bounded.bound/bind})\quad \frac{\mathsf{isBounded}\,(A; e; c) \qquad \forall a : A.\, \mathsf{isBounded}\,(B; f(a); d(a))}{\mathsf{isBounded}\,(B; \mathsf{bind}(e; f); \mathsf{bind}(e; \lambda a.\, c + d(a)))}$$

$$\textsc{Relax}\ (\texttt{Calf.Types.Bounded.bound/relax})\quad \frac{\mathsf{isBounded}\,(A; e; c) \qquad c \leq c'}{\mathsf{isBounded}\,(A; e; c')}$$

Fig. 4. Quantitative refinement lemmas in **calf** displayed in inference rule style.

clocked program to satisfy the behavioral specification of the algorithm. By definition the *recursion depth* of the program is a safe instantiation.

Furthermore, note that the *cost* of the program is an upper bound on the recursion depth in many cost models. In such cases defining an algorithm in **calf** is intertwined with extracting and verifying its cost bound, evincing a synergy one enjoys in the cost-aware setting: algorithms with interesting cost structure require general recursive definitions, meanwhile their safety as clocked programs is derived from the cost bound. Observe that this paradigm is a legitimate encoding of general recursion because *we do not track the cost of computing the cost bound.* One may think of this arrangement as programming with a version of for loops whose bounds are computed in a cost-free manner.

Method of recurrence relations. To put the plan in action, we outline a recipe for defining and analyzing an algorithm using the method of recurrence relations in **calf**:

(1) An algorithm is given along with its *cost model*. Place step in accordance with the cost model to obtain a cost-aware instrumentation of the algorithm.
(2) Define a *clocked* version of the algorithm; explicitly, one parametrizes the algorithm by an extra clock argument of type nat representing the available fuel; when the clock is nonzero, the program follows the designated recursion pattern by decrementing the clock, and when the clock is zero, the program terminates by returning a default value or raising an exception.
(3) Define the *recursion depth* that bounds the number of recursive calls. Because we do not track the cost of computing the recursion depth, it may be defined however convenient.
(4) Define the the associated *cost recurrence* that maps inputs and to costs. Often times this may be used in place of the recursion depth as it is an upper bound. Similar to the recursion depth, we do not track the cost of the cost recurrence.
(5) Obtain the *complete program* by instantiating the clocked program with the recursion depth. Prove this is a safe instantiation in the sense that the resulting program satisfies the behavioral specification of the algorithm (*e.g.* computes the greatest common divisor).
(6) Prove that the resulting algorithm is bounded by the cost recurrence. This process is mostly mechanical: one repeatedly applies the lemmas in Section 3 to break down isBounded goals.
(7) Refine the recurrence by (*e.g.*) computing a closed-form solution. Usually this step represents the bulk of the work in pen-and-paper algorithm analysis.

We apply this recipe in the following section to analyze Euclid's algorithm for the greatest common divisor.

4 VERIFICATION IN calf

We demonstrate in **calf** two fundamental techniques used pervasively in algorithm analysis. First, we illustrate the method of recurrence relations by analyzing Euclid's algorithm for the greatest common divisor, proving its correctness and deriving an asymptotically tight upper bound on the number of modulus operations used. Second, we formalize the physicist's method for amortized analysis by studying the complexity of sequences of *batched queue* operations, verifying that each queue operation has constant amortized cost. Due to space limitations we cannot discuss the sorting case study in detail, but we mention results concerning parallel complexity in Section 6, and the interested reader can find the full development in the Agda formalization.

Through these case studies, we promote a comprehensive verification pipeline made possible by the unification of the following ingredients in a single framework:

(1) Specification of cost models
(2) Formal connection between algorithms and their associated recurrence relations
(3) A modality that administers extensional properties
(4) Full-spectrum dependent types that provides a rich specification language

Cost models. Prior to analyzing an algorithm, one has to make clear what "counts" as cost. A particularly simple definition is to count every transition step in an operational semantics, resulting in a *language-level* cost semantics. On the other hand, algorithms researchers prefer a different perspective in which cost is an *algorithm-specific* notion. For example, a common cost model for sorting algorithms counts the number of comparisons, which does not account for the cost of (*e.g.*) constructing lists. This view allows one to study the underlying combinatorial structure of an algorithm without getting distracted by implementation details. This is the prevailing perspective we take in **calf**, although one can also work with a uniform language-level cost semantics when necessary; for instance, in the amortized analysis of batched queues (see Section 4.2) we axiomatize a type of *cost-aware* lists that charges one step per recursive call.

Formalizing recurrence extraction. Recurrence relations are a fundamental concept in algorithm analysis — every algorithm can be *abstracted* into an associated cost recurrence that characterizes the relationship between the input and the induced cost. Recent work of Kavvos et al. [2019] has provided mathematical grounding for informal proofs involving recurrence relations in the form of a verified procedure for extracting (higher-order) recurrence relations from CBPV programs. Although **calf** does not support recurrence extraction in the mechanical style proposed by Kavvos et al. [2019], one can manually define a recurrence and express its relationship to the given algorithm by proving the *internal* isBounded refinement. Indeed, one of the advances embodied in **calf** is the unification of the distinct phases/languages in Kavvos et al. [2019] into a single framework that furnishes a programming language with support for cost specification.

Managing extensionality. As discussed in Section 1.3, the language of phase distinctions naturally induces a modality $\bigcirc$ for extension, which we use to express behavioral specifications in **calf**. For instance, we express the correctness of Euclid's algorithm by proving that it satisfies the characteristic equations of the gcd under the extensional modality $\bigcirc$.

Cost-aware logical framework. Decades of experience has shown the effectiveness of using dependent type theories to encode mathematics [Buzzard et al. 2020; Gonthier 2008; Han and van Doorn 2020] and to verify *behavioral* properties of programs [Chlipala 2013; Lee et al. 2007; Stump 2016; Ullrich 2016]. Our experience with **calf** suggests that dependent type theories are *also* an appropriate tool for analyzing intensional properties of programs including cost. In the following case studies we rely on the rich type structure of **calf** to evaluate different strategies for establishing

$$mod_{\mathsf{inst}} : \mathsf{tm}^{+}(\mathsf{nat}) \to \mathsf{tm}^{+}(\mathsf{nat}) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{nat}))$$
$$mod_{\mathsf{inst}}(x, y) = \mathsf{step}^{1}(\mathsf{ret}(mod\,(x, y)))$$

$$gcd_{\mathsf{clocked}} : \mathsf{tm}^{+}(\mathsf{nat}) \to \mathsf{tm}^{+}(\mathsf{nat}^{2}) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{nat}))$$
$$gcd_{\mathsf{clocked}}\,(\mathsf{zero})\,(x, y) = \mathsf{ret}(x)$$
$$gcd_{\mathsf{clocked}}\,(\mathsf{suc}(k))\,(x, \mathsf{zero}) = \mathsf{ret}(x)$$
$$gcd_{\mathsf{clocked}}\,(\mathsf{suc}(k))\,(x, \mathsf{suc}(y)) = \mathsf{bind}(mod_{\mathsf{inst}}(x, \mathsf{suc}(y)); \lambda r.\, gcd_{\mathsf{clocked}}(k)(\mathsf{suc}(y), r))$$

$$gcd_{\mathsf{depth}} : \mathsf{tm}^{+}(\mathsf{nat}^{2}) \to \mathsf{tm}^{+}(\mathsf{nat})$$
$$gcd_{\mathsf{depth}}(x, y) = \begin{cases} \mathsf{zero} & \text{if y = zero} \\ \mathsf{suc}(gcd_{\mathsf{depth}}(y, mod\,(x, y))) & \text{o.w.} \end{cases}$$

$$gcd : \mathsf{tm}^{+}(\mathsf{nat}^{2}) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{nat}))$$
$$gcd\,(x, y) = gcd_{\mathsf{clocked}}\,(gcd_{\mathsf{depth}}\,(x, y))(x, y)$$

Fig. 5. Euclid's algorithm in **calf**. From top to bottom: mod_{inst} is the cost instrumented modulus operation, gcd_{clocked} is the clocked algorithm, gcd_{depth} is the recursion depth/cost recurrence, and gcd is the final program. Note that because $gcd_{\mathsf{depth}}$is cost-free, we may define it however convenient, *e.g.* by well-founded induction on the arguments.

cost bounds. We emphasize that **calf** is a *framework* for quantitative reasoning: instead of working with a fixed set of rules, one is free to choose the most appropriate tool for the given problem.

4.1 Euclid's Algorithm

In our first case study we analyze Euclid's algorithm for calculating the greatest common divisor, the prototypical example of an algorithm that relies on nonstructural recursion. Our analysis closely follows the steps in the recipe from Section 3.2.

Behavioral specification. Let $gcd : \mathsf{tm}^{+}(\mathsf{nat}^{2}) \to \mathsf{tm}^{\ominus}(\mathsf{nat})$ be a candidate **calf** program for computing the gcd. Inspired by the usual formulation of Euclid's algorithm, we may specify the correct *behavior* of gcd with the following propositions:

$$\bigcirc(gcd\,(x, \mathsf{zero}) = \mathsf{ret}(x)) \tag{3}$$
$$\bigcirc(gcd\,(x, \mathsf{suc}(y)) = gcd\,(\mathsf{suc}(y), mod\,(x, \mathsf{suc}(y)))) \tag{4}$$

Above we have assumed that there is a (cost-free) **calf** program $mod : \mathsf{tm}^{+}(\mathsf{nat}^{2}) \to \mathsf{tm}^{+}(\mathsf{nat})$ that computes the modulus. In other words Eqs. (3) and (4) state that gcd satisfies the defining clauses of Euclid's algorithm in the extensional fragment.

Specializing the cost structure. Because the gcd is defined on the natural numbers, we instantiate the cost structure $\mathbb{C}$ at the ordered monoid $(\mathbb{N}, +, 0, \leq)$.

Executing the recipe. We execute the recipe from Section 3.2 to analyze Euclid's algorithm. The associated **calf** programs are displayed in Fig. 5. First we define the cost model to be the number of *mod* operations, encoded in the instrumented version of the modulus, mod_{inst}, which is used to define the clocked gcd algorithm gcd_{clocked}. Here the first parameter of gcd_{clocked} serves as the termination metric: recursive calls in Euclid's algorithm are justified by decrementing the clock parameter. Next, observe that under our cost model the recursion depth and cost recurrence coincide for

Euclid's algorithm, so we define a single (cost-free) program gcd_{depth} that simultaneously provides a sufficient instantiation (described in Section 3.2) of the clock in gcd_{clocked} and a cost recurrence for the algorithm. Consequently the complete algorithm gcd is obtained by instantiating the clock parameter in gcd_{clocked} with gcd_{depth}. We prove that gcd correctly implements gcd:

Theorem 4.1 (`Examples.Gcd.Spec.{gcd≡spec/zero, gcd≡spec/suc}`). *We have that gcd behaves correctly,* i.e. *Eqs.* (3) *and* (4) *hold for all* $x, y : \mathsf{tm}^+(\mathsf{nat})$.

Let ι be the obvious isomorphism $\mathsf{tm}^+(\mathsf{nat}) \cong \mathbb{N}$.[4] We verify that gcd is bounded by $\iota \circ gcd_{\mathsf{depth}}$:

Theorem 4.2 (`Examples.Gcd.Clocked.gcd≤gcd/depth`). *For all* $x, y : \mathsf{tm}^+(\mathsf{nat})$, *we have that* $\mathsf{isBounded}\left(\mathsf{nat}; gcd\,(x, y); (\iota \circ gcd_{\mathsf{depth}})(x, y)\right)$.

Lastly we prove a refinement for the recurrence gcd_{depth} by computing a closed-form bound. Let $\mathsf{Fib} : \mathbb{N} \to \mathbb{N}$ be the fibonacci sequence, and let $\mathsf{Fib}^{-1} : \mathbb{N} \to \mathbb{N}$ be the function characterized by the equation $\mathsf{Fib}^{-1}(x) = \max\{i \mid \mathsf{Fib}(i) \leq x\}$. Note that Fib^{-1} is well-defined since Fib is strictly monotonic for $n \geq 2$. It is well-known that the cost bound $\iota \circ gcd_{\mathsf{depth}}$ is closely related to Fib^{-1}:

Theorem 4.3 (`Examples.Gcd.Refine.gcd/cost≤gcd/depth/closed`). *For all* $x, y : \mathsf{tm}^+(\mathsf{nat})$, *we have that* $(\iota \circ gcd_{\mathsf{depth}})(x, y) \leq \mathsf{Fib}^{-1}(\iota(x)) + 1$.

Corollary 4.4 (`Examples.Gcd.Refine.gcd≤gcd/depth/closed`). *For all* $x, y : \mathsf{tm}^+(\mathsf{nat})$, *we have that* $\mathsf{isBounded}\left(\mathsf{nat}; gcd\,(x, y); \mathsf{Fib}^{-1}(\iota(x)) + 1\right)$.

4.2 Amortized Analysis

In addition to the method of recurrence relations, we may formulate more advanced algorithm analysis techniques. As an example, we illustrate the **calf** formalization of *amortized analysis*. First introduced by Tarjan in the mid-80s, amortized analysis is a method to establish cost bounds on *sequences* of operations on a data structure that is more precise than a simple union bound. In this section we present a version of amortized analysis known as the physicist's method: given a data structure s, one may define a measure $\Phi : s \to \mathbb{Z}_+$ that represents the amount of *potential* that can be used to do work. The crux of the analysis is to rig Φ so that expensive operations are associated with large decreases in potential; because Φ is nonnegative, this ensures that expensive operations cannot occur too often in a given sequence, *i.e.* their cost is *amortized*.

Batched queues. To illustrate the physicist's method, we analyze the amortized complexity of a queue implementation known as *batched queues* [Burton 1982; Gries 1987; Hood and Melville 1981; Okasaki 1998]. A batched queue is a pair of lists (f, b) coupled with the invariant that the logical order of the queue is $f :: rev(b)$. The **calf** implementation of the batched queue is presented in Fig. 6. For simplicity, we only consider elements of type nat.

Specializing the cost structure. For amortized analysis of batched queues, we instantiate the cost monoid $\mathbb{C}$ at the ordered monoid $(\mathbb{N}, +, 0, \leq)$ whose structure as a semiring and compatibility with the integers $\mathbb{Z}$ are required to define and reason about the potential function.

Cost model. A common cost model in this setting is the number of list iterations. We encode this cost model by axiomatizing a type of cost-aware lists $\mathsf{L} : \mathbb{C} \to \mathsf{tp}^+ \to \mathsf{tp}^+$. that is parameterized by the amount to charge for each recursive call. The type L has the standard constructors nil and cons;

[4] Because both the cost monoid $\mathbb{N}$ and nat are defined via the Agda natural numbers, ι is the identity in our implementation.

$$Q \coloneqq \mathsf{L}^1(\mathsf{nat}) \times \mathsf{L}^1(\mathsf{nat}) \qquad enq : \mathsf{tm}^+(Q) \to \mathsf{tm}^+(\mathsf{nat}) \to \mathsf{tm}^+(Q)$$
$$enq((f,b),x) = (f, \mathsf{cons}(x;b))$$

$$deq_0 : \mathsf{tm}^+(\mathsf{list}(\mathsf{nat})) \to \mathsf{tm}^\ominus(\mathsf{F}(1 + Q \times \mathsf{nat}))$$
$$deq_0(b) = l \leftarrow rev(b); \mathsf{rec}_\mathsf{L}(l)\{\mathsf{ret}(\mathsf{inl}(\star)) \mid \lambda a, l', \_.\, \mathsf{ret}(\mathsf{inr}((l', \mathsf{nil}), a))\}$$

$$deq : \mathsf{tm}^+(Q) \to \mathsf{tm}^\ominus(\mathsf{F}(1 + (Q \times \mathsf{nat})))$$
$$deq((f,b)) = \mathsf{rec}_\mathsf{L}(f)\{deq_0(b) \mid \lambda a, f', \_.\, \mathsf{ret}(\mathsf{inr}((f', b), a))\}$$

Fig. 6. Batched queues in **calf**.

the only new rule is the destruction of cons nodes, which induces the annotated amount of cost:

$$\mathsf{rec/cons} : \{c, A, a, X, e_0, e_1\}\ (l : \mathsf{tm}^+(\mathsf{L}^c(A))) \to$$
$$\mathsf{rec}_\mathsf{L}(\mathsf{cons}(a;l); X; e_0; e_1) = \mathsf{step}^c(e_1(a)(l)(\mathsf{rec}_\mathsf{L}(l; X; e_0; e_1)))$$

To charge unit cost per iteration, we define the type of batched queues as $Q \coloneqq \mathsf{L}^1(\mathsf{nat}) \times \mathsf{L}^1(\mathsf{nat})$. Note that the standard list type is recovered as $\mathsf{list}(A) \coloneqq \mathsf{L}^0(A)$. We write $|-| : \{c\}\ \mathsf{L}^c(A) \to \mathbb{N}$ for the length function on lists.

Upper bounding individual queue operations. We obtain cost bounds on the individual operations using similar techniques as in Section 4.1:

Theorem 4.5 (`Examples.Queue.enq≤enq/cost`, `Examples.Queue.deq≤deq/cost`). *For any queue q and element x, we have* $\mathsf{isBounded}\,(Q; enq(q,x); 0)$. *Moreover, for any queue $q = (f,b)$, we have* $\mathsf{isBounded}\,(1 + Q \times \mathsf{nat}; deq(q); 1 + |b|)$.

Serializing the queue operations. To formalize the notion of a sequence of operations, we define a serialization of the queue operations in Fig. 7. Here, op denotes the type of queue operations, which is either an enqueue of an element or a dequeue. Given a serialized operation o and a queue q, $[\![o]\!](q)$ is the interpretation of o on q. By Theorem 4.5 the resulting computation is bounded by the cost of the corresponding operation $cost(q,o)$, defined in Fig. 7:

Corollary 4.6 (`Examples.Queue.op≤op/cost`). *Given an operation o and a queue q, we have* $\mathsf{isBounded}\,\big(Q; [\![o]\!](q); cost(q,o)\big)$.

The function $[\![-]\!]_{\mathsf{seq}}(-)$ lifts the interpretation to sequences of operations by threading the given queue through the list of operations. It is bounded by $cost_{\mathsf{seq}}$:

Lemma 4.7 (`Examples.Queue.op/seq≤op/seq/cost`). *Given a list of operations l and a queue q, we have* $\mathsf{isBounded}\,\big(Q; [\![l]\!]_{\mathsf{seq}}(q); cost_{\mathsf{seq}}(l,q)\big)$.

Amortized analysis of batched queues. We are now in a position to analyze the amortized cost of the queue operations. We define the potential function on queue states:

$$\Phi : \mathsf{tm}^+(Q) \to \mathbb{N}$$
$$\Phi(f,b) = |f| + 2 \cdot |b|$$

$$\begin{aligned}
\mathsf{op} &: \mathsf{tp}^+ & \mathit{cost} &: \mathsf{tm}^+(\mathsf{op}) \to \mathsf{tm}^+(Q) \to \mathbb{Z} \\
\mathsf{op} &= \mathsf{nat} + 1 & \mathit{cost}(\mathit{op}_{enq}(x), q) &= 0 \\
\mathit{op}_{enq}(x) &= \mathsf{inl}(x) & \mathit{cost}(\mathit{op}_{deq}, (f, b)) &= 1 + |b| \\
\mathit{op}_{deq} &= \mathsf{inr}(\star)
\end{aligned}$$

$$\begin{aligned}
&[\![-]\!](-) : \mathsf{tm}^{\ominus}(\mathsf{op} \to Q \to \mathsf{F}(Q)) \\
&[\![\mathit{op}_{enq}(x)]\!](q) = \mathit{enq}(q, x) \qquad [\![-]\!]_{\mathsf{seq}}(-) : \mathsf{tm}^{\ominus}(\mathsf{list}(\mathsf{op}) \to Q \to \mathsf{F}(Q)) \\
&[\![\mathit{op}_{deq}]\!](q) = s \leftarrow \mathit{deq}(q); \qquad [\![\mathsf{nil}]\!]_{\mathsf{seq}}(q) = \mathsf{ret}(q) \\
&\quad \mathsf{case}(s)\{\ \mathsf{inl}(\star) \hookrightarrow \mathsf{ret}((\mathsf{nil}, \mathsf{nil})) \quad [\![\mathsf{cons}(o; os)]\!]_{\mathsf{seq}}(q) = q' \leftarrow [\![o]\!](q); f(q') \\
&\qquad \mid \mathsf{inr}((q, x)) \hookrightarrow \mathsf{ret}(q)\ \}
\end{aligned}$$

$$\begin{aligned}
\mathit{cost}_{\mathsf{seq}} &: \mathsf{tm}^+(\mathsf{list}(\mathsf{op})) \to \mathsf{tm}^+(Q) \to \mathbb{Z} \\
\mathit{cost}_{\mathsf{seq}}(\mathsf{nil}, q) &= 0 \\
\mathit{cost}_{\mathsf{seq}}(\mathsf{cons}(o; os), q) &= \mathit{cost}(o, q) + (q' \leftarrow [\![o]\!](q); \mathit{cost}_{\mathsf{seq}}(os, q'))
\end{aligned}$$

Fig. 7. Serialization of queue operations.

Traditionally an operation's amortized cost is defined as the maximum value of the sum of the induced cost and the difference in the potential over a starting state; we represent this relationally:

$$\begin{aligned}
\mathsf{hasCost}_{\mathsf{amortized}} &: \mathsf{tm}^+(\mathsf{op}) \to \mathbb{N} \to \mathbf{Jdg} \\
\mathsf{hasCost}_{\mathsf{amortized}}(o, k) &= (q : Q) \to \left(\mathit{cost}(o, q) +_{\mathbb{Z}} \Phi([\![o]\!](q)) -_{\mathbb{Z}} \Phi(q)\right) \leq_{\mathbb{Z}} k
\end{aligned}$$

Note that because amortized cost has to be defined using *non-truncated* subtraction, terms of type $\mathbb{N}$ appearing in the relation $\mathsf{hasCost}_{\mathsf{amortized}}$ are all implicitly lifted to the integers $\mathbb{Z}$. We verify that the amortized cost of enqueue is 2, while the amortized cost of dequeue is 0:

Theorem 4.8 (`Examples.Queue.enq/acost`, `Examples.Queue.deq/acost`). *We have that* $\mathsf{hasCost}_{\mathsf{amortized}}(\mathit{op}_{enq}(x), 2)$ *for all* $x : \mathsf{tm}^+(\mathsf{nat})$ *and that* $\mathsf{hasCost}_{\mathsf{amortized}}(\mathit{op}_{deq}, 0)$.

Using the amortized costs, we can bound the cost of a sequence of queue operations using a standard telescoping series:

Theorem 4.9 (`Examples.Queue.op/seq/cost≤ϕ₀+2*|l|`). *Given an initial queue* $q : \mathsf{tm}^+(Q)$ *and a list of operations* $l : \mathsf{tm}^+(\mathsf{list}(\mathsf{op}))$, *we have* $\mathit{cost}_{\mathsf{seq}}(l, q) \leq \Phi(q) + 2|l|$.

Combining this inequality with Lemma 4.7, we obtain an amortized bound on a sequence of operations on the empty queue:

Corollary 4.10 (`Examples.Queue.op/seq≤2*|l|`). *Given a list of operations* l, *we have that* $\mathsf{isBounded}\left(Q; [\![l]\!]_{\mathsf{seq}}((\mathsf{nil}, \mathsf{nil})); 2|l|\right)$.

5 METATHEORY OF calf

In this section we substantiate the theory of **calf** by means of a model construction and prove the following theorems:

(1) ***Nondegeneracy.*** The cost effect step is not degenerate, i.e $\not\vdash \mathsf{step}^1(e) = e$ for any $e : \mathsf{F}(A)$.
(2) ***Validity of cost bounds.*** We have that $\vDash \bigcirc(m \leq n)$ if and only if $\vDash m \leq n$ for all $m, n : \mathbb{N}$.

*Models of **calf**.* Recall from Section 2.8 that we define **calf** as the *free* lccc $\mathscr{C}_{\mathbf{calf}}$ over the signature $\Sigma_{\mathbf{calf}}$ presented in Fig. 3. Consequently one may prove metatheorems about **calf** using the universal property of freely generated categories. In the context of functorial semantics [Lawvere 1963], the universal property states that one may define a model $\mathscr{C}_{\mathbf{calf}} \to \mathscr{E}$ by simply specifying the image of the constants of $\Sigma_{\mathbf{calf}}$ in $\mathscr{E}$ [5]. The data of this specification is encapsulated by the notion of an *algebra* for a signature:

Definition 5.1 (Algebra for a signature in the logical framework). Let $\mathscr{E}$ be a category that has a universe $\mathcal{U}$ closed under dependent products, dependent sums, and extensional equality. Given a signature Σ in the logical framework, we can define a type $\mathbf{Alg}_{\mathcal{U}}(\Sigma)$ of $\mathcal{U}$-small algebras for Σ in $\mathscr{E}$ by interpreting **Jdg** as $\mathcal{U}$ and taking the dependent sum over all the constants declared in Σ.

Thus given a sufficiently structured category $\mathscr{E}$ in the sense above, we can define a model of **calf** by exhibiting an algebra $\mathcal{A} : \mathbf{Alg}_{\mathcal{U}}(\Sigma_{\mathbf{calf}})$ in some universe $\mathcal{U}$ of $\mathscr{E}$. In fact we can define the intended model of **calf** in *any* given topos $\mathbf{X}$ with a distinguished subterminal object representing the phase separation of intension and extension. To obtain an external view, we specialize the construction to the presheaf topos over the interval category $\{0 \to 1\}$, *i.e.* the category of families of sets $\mathbf{Set}^{\to}$, which suggests the interpretation of **calf** types as phase separated *families*.

Language of phase distinctions. Inspired by recent work emphasizing the role of phase distinctions in the analysis of metatheoretic properties [Sterling and Angiuli 2021; Sterling and Harper 2021], we isolate a pair of complementary modalities $\bigcirc, \bullet$ that models the phase distinction of extension and intension in **calf**. Using the language of phase distinctions, we give a succinct definition of our model that avoids the explicit but more cumbersome presentation involving families.

5.1 Counting Model of calf

We exhibit an algebra $\mathcal{A}$ for $\Sigma_{\mathbf{calf}}$ in any given topos $\mathbf{X}$ equipped with a distinguished proposition $¶_{\mathsf{E}} : \Omega$.[6] Consequently we have at our disposal a rich internal language in the form of an extensional dependent type theory that includes (in particular) a hierarchy of universes $\mathcal{U}_\alpha$, inductive types, and a universe of proof-irrelevant propositions Ω. The role of the proposition $¶_{\mathsf{E}}$ is to provide a semantic counterpart to the **calf** proposition $¶_{\mathsf{E}}$.

Letting $\alpha < \beta$ be universe levels, we then define an algebra $\mathcal{A} : \mathbf{Alg}_{\mathcal{U}_\beta}(\Sigma_{\mathbf{calf}})$ that constitutes the standard Eilenberg–Moore model of CBPV in which computation types are interpreted as *algebras* for a given monad.[7] In the case of **calf** we dub this interpretation the *counting model*, so named because the interpretation of the computation type $\mathsf{F}(A)$ is the free algebra of a particular writer monad whose carrier classifies elements of A paired with a step count. Because many parts of the interpretation are standard, we highlight only the constructions pertaining to **calf** per se.

5.1.1 Phase Distinction. As mentioned above, we define the extensional phase $¶_{\mathsf{E}}$ as the distinguished proposition $¶_{\mathsf{E}}$. By definition, the extensional modality is rendered as the function space in the internal language of $\mathbf{X}$, i.e $\bigcirc - := ¶_{\mathsf{E}} \to -$. The intensional modality $\bullet -$ is defined as the pushout $A \sqcup_{A \times ¶_{\mathsf{E}}} ¶_{\mathsf{E}}$ of the projection maps of $A \times ¶_{\mathsf{E}}$.

Proposition 5.2 (Rijke et al. [2020]). *Both $\bigcirc, \bullet$ are idempotent, left exact and monadic.*

We write $(\eta_{\bigcirc}, \eta_{\bullet})$ for the monadic unit of the (extensional, intensional) modality. Observe that $\bullet A$ collapses to a single point when a proof of $¶_{\mathsf{E}}$ exists:

[5] An analogous situation arises when considering homomorphisms out of a free group: *any* function on the generators determines a homomorphism.

[6] For the limited topos theory we require in this section, we employ the notations of Anel and Joyal [2021].

[7] Not to be confused with Definition 5.1.

Proposition 5.3. *Given $u : \P_{\mathsf{E}}$, we have that $\bullet A \cong 1$ for any A.*

Thus we may effect the erasure of step in the extensional fragment by arranging the cost structure of programs to be a type in the *image* of $\bullet$: when a proof $u : \P_{\mathsf{E}}$ is present, a cost c is equal to any other cost, in particular 0; consequently we have $\mathsf{step}^c(e) = \mathsf{step}^0(e) = e$ by the coherence of step.

5.1.2 Cost Monoid $\mathbb{C}$. Recalling that **calf** is parameterized in a cost monoid $\mathbb{C}$, our model takes as an input an arbitrary $(\mathbb{M}, +, 0, \leq)$ cost monoid in the category of sets **Set**. We then define $\mathbb{C}$ as the image of $\mathbb{M}$ under the constant sheaf functor $\mathbf{Set} \to \mathrm{Sh}(\mathbf{X})$. Note that because $\mathbb{C}$ is not necessarily in the image of $\bullet$, we interpret computation types of **calf** as algebras for the writer monad $\bullet\mathbb{C} \times -$. By Proposition 5.3 the cost structure of programs is then rendered trivial underneath $\P_{\mathsf{E}}$.

5.1.3 Judgmental Structure. Per the Eilenberg–Moore model of CBPV, value types **calf** are simply interpreted as types in $\mathbf{X}$, and computation types are interpreted as algebras for $\bullet\mathbb{C} \times -$:

$$\mathsf{alg}(T) = \begin{cases} \mathsf{A} : \mathcal{U}_\alpha & \mathsf{tp}^+ : \mathcal{U}_\beta \qquad \mathsf{tp}^\ominus : \mathcal{U}_\beta \\ \mathsf{map} : T(\mathsf{A}) \to \mathsf{A} & \mathsf{tp}^+ = \mathcal{U}_\alpha \qquad \mathsf{tp}^\ominus = \mathsf{alg}(\bullet\mathbb{C} \times -) \\ \mathsf{unit} : \mathsf{map} \circ \eta = id_{\mathsf{A}} & \mathsf{tm}^+(A) = A \qquad \mathsf{tm}^\ominus(X) = |X| \\ \mathsf{mult} : \mathsf{map} \circ \mu = \mathsf{map} \circ T\mathsf{map} \end{cases}$$

Note that given an algebra X, we write $|X|$ for the carrier $X \cdot \mathsf{A}$.

5.1.4 Values and Computations. In the algebra semantics of CBPV, one coerces between value types and computation types via the adjoint pair $\mathsf{F} \dashv \mathsf{U}$ in which the left adjoint takes a type to the associated free $\bullet\mathbb{C} \times -$-algebra and the right adjoint forgets the structure of the given algebra, writing $\mathsf{freeAlg}(T, A)$ for the free T algebra on A:

$$\mathsf{F} : \mathcal{U}_\alpha \to \mathsf{alg}(\mathbb{C} \times -) \qquad \mathsf{U} : \mathsf{alg}(\bullet\mathbb{C} \times -) \to \mathcal{U}_\alpha$$
$$\mathsf{F}(A) = \mathsf{freeAlg}(\bullet\mathbb{C} \times -, A) \qquad \mathsf{U}(X) = |X|$$

5.1.5 Cost Effect. The cost effect step is given by the algebra map of the given computation type:

$$\mathsf{step} : \{X\}\ \mathbb{C} \to |X| \to |X|$$
$$\mathsf{step}^c(x) = (X \cdot \mathsf{map})(\eta_\bullet(c), x)$$

The following is an immediate consequence of Proposition 5.3.

Corollary 5.4 (Extensional fragment). *We have that $\bigcirc(\mathsf{step}^c(e) = e)$ for all $c : \mathbb{C}$ and $e : |X|$.*

Counting model in $\boldsymbol{Set}^{\to}$. We may obtain a more concrete perspective on the counting model $\mathcal{A}$ by considering its construction in the arrow category $\mathbf{Set}^{\to}$ in which the extensional phase $\P_{\mathsf{E}}$ is furnished by the subterminal family $0 \to 1$. Observe that objects in this category are families of sets $A : A_1 \to A_0$, which corresponds to the fact that a type A is a family indexed in a collection of *behaviors* with the fibers representing the *cost structure* for a given behavior.

In $\mathbf{Set}^{\to}$ the extensional modality takes a family $A_1 \to A_0$ to the identity $A_0 \to A_0$, trivializing the fiber (*i.e.* cost structure) over each point in A_0. On the other hand, the intensional modality takes $A_1 \to A_0$ to the family $A_1 \to 1$; applying the extensional modality thence results in the terminal family $1 \to 1$, illustrating the fact that the extensional part of the intensional part of any type is trivial.

5.2 Nondegeneracy of step

Theorem 5.5. *We have that* $(\mathsf{step}^c(e) = e) \to \bullet\bot$ *for any nonzero* $c : \mathbb{C}$ *and* $e : \bullet\mathbb{C} \times A$.

Proof. By definition, $e = (c', a)$ for some $c' : \bullet\mathbb{C}$ and $a : A$. Unfolding the definition of step and free algebra, we have $\mathsf{step}^c(c', a) = (\eta_\bullet(c) +_\bullet c', a)$, where $+_\bullet$ lifts $+$ using the functorial action of $\bullet$. Hence it suffices to show $(\eta_\bullet(c) +_\bullet c', a) = (c', a)$ implies $\bullet\bot$. Suppose $(\eta_\bullet(c) +_\bullet c', a) = (c', a)$. By the induction principle of pushouts, there are two cases to consider. First, suppose $c' = \eta_\bullet(c'')$ for some $c'' : \mathbb{C}$. Because $\bullet$ is left exact, the equation $\eta_\bullet(c) +_\bullet \eta_\bullet(c'') = \eta_\bullet(c'')$ is equivalent to $\bullet(c + c'' = c'')$. But we assumed that c is nonzero, so the fact that $\mathbb{C}$ is cancellative entails $c + c'' = c'' \to \bot$, and the result follows from the functorial action of $\bullet$. On the other hand, suppose $c' = *(u)$ for some $u : \P_{\mathsf{E}}$. By Lemma 5.3, we obtain a unique proof of $\bullet\bot$. □

Because $\bullet\bot = \P_{\mathsf{E}}$, we know that if step is degenerate, then the extensional phase $\P_{\mathsf{E}}$ is derivable. Observing that we placed no restrictions on the proposition $\P_{\mathsf{E}}$ in the construction of the counting model, we immediately obtain the desired theorem by instantiating $\P_{\mathsf{E}}$ with the false proposition:

Theorem 5.6. *We have that* $\nvdash \mathsf{step}^c(e) = e$ *for any nonzero* $c : \mathbb{C}$ *and* $e : \mathsf{F}(A)$.

5.3 Validity of Extensional Cost Bounds

We show that extensional inequalities are equivalent to ordinary inequalities in $\mathcal{A}_{\mathbf{Set}^{\to}}$, the $\mathbf{Set}^{\to}$ model of **calf**, whenever the cost monoid is *extensional* in the sense that $\mathbb{C} \cong \bigcirc\mathbb{C}$ and the relation $\leq$ may be characterized using Σ and equality types. As an example, we illustrate the case for the cost monoid $\mathbb{N}$, noting that the same proof may be easily adapted to other common cost monoids:

Theorem 5.7. *We have that* $\mathcal{A}_{\mathbf{Set}^{\to}} \vDash \bigcirc(m \leq n)$ *if and only if* $\mathcal{A}_{\mathbf{Set}^{\to}} \vDash m \leq n$ *for all* $m, n : \mathbb{N}$.

Proof. Observe that we may present $m \leq n$ as the type $\Sigma k : \mathbb{N}.\, n = \mathsf{suc}^k(m)$. By standard results [Rijke et al. 2020] we know that the property of being extensional is closed under equality and Σ types. Combined with the fact that $\mathbb{N}$ is an extensional type in the $\mathbf{Set}^{\to}$ model of **calf**, we conclude that $m \leq n$ is also extensional, *i.e.* $(m \leq n) \cong \bigcirc(m \leq n)$. □

It may be natural to ask if Theorem 5.7 holds for the syntactic model of **calf**, *i.e.* is it the case that $\vdash \bigcirc(m \leq n)$ if and only if $\vdash m \leq n$. While the backwards implication is immediate from the definition of the extensional modality, the forward implication requires the fact that canonicity holds for **calf** (so that any closed term $\vdash n : \mathbb{N}$ is equal to a numeral). We conjecture that the techniques of synthetic Tait computability developed by Sterling and Harper [2021] can be used in the setting of **calf** to give a succinct proof of canonicity, but we do not claim any technical results.

Alternatively, one may axiomatize a version of **calf** with a constant of type $\bigcirc(m \leq n) \to m \leq n$ so that extensional inequality and inequality are equi-derivable internally. Executing this approach requires one to restrict to cost monoids that are extensional and whose ordering relation is definable using Σ and equality types. To keep the interpretation of **calf** more open-ended, we simply observe via Theorem 5.7 that a large class of models of **calf** validates extensional cost bounds.

6 PARALLELISM IN calf

Parallelism arises naturally in the setting of **calf** via an equational presentation of the profiling semantics of Blelloch and Greiner [1995]. Here we present a version adapted from Harper [2018] in which it is observed that the source of parallelism can be isolated to the treatment of *pairs* of computations: a parallel computation of $A \times B$ is furnished by a new computation form $\&$ that conjoins two independent computations of A and B:

$$\& : \{A, B : \mathsf{tp}^+\}\ \mathsf{tm}^{\ominus}(\mathsf{F}(A)) \to \mathsf{tm}^{\ominus}(\mathsf{F}(B)) \to \mathsf{tm}^{\ominus}(\mathsf{F}(A \times B))$$

One may think of a term $e \mathbin{\&} f$ as a computation in which e and f are evaluated simultaneously.

Cost structure of parallelism. Blelloch and Greiner [1995] characterize the complexity of a program in terms of two measures: *work*, which represents its sequential cost, and *span*, which represents its parallel cost. In **calf** this structure is recorded by the *parallel cost monoid* $\mathbb{C} \coloneqq (\mathbb{N}^2, \oplus, (0,0), \leq_{\mathbb{N}^2})$ in which $\oplus$ and $\leq_{\mathbb{N}^2}$ are component-wise extensions of addition and $\leq$. Parallel cost composition is then implemented by the operation $(w_1, s_1) \otimes (w_2, s_2) \coloneqq (w_1 + w_2, \max(s_1, s_2))$ that takes the sum of the works and max of the spans. This provides the required structure to assemble the cost of a completed parallel pair:

$$\&_{\mathsf{join}} : \{A, B, c_1, c_2, a, b\}\ (\mathsf{step}^{c_1}(\mathsf{ret}(a))) \mathbin{\&} (\mathsf{step}^{c_2}(\mathsf{ret}(b))) = \mathsf{step}^{c_1 \otimes c_2}(\mathsf{ret}((a, b)))$$

*Nondegeneracy of parallel **calf**.* Metatheoretic properties of parallel **calf** follows directly from the counting model defined in Section 5, given that we can interpret parallel pairing. Because the new pairing operation is only defined on free algebras, we may use $\otimes_{\bullet}$ (lift of $\otimes$ by the functorial action of $\bullet$) to define parallel pairing: $(c_1, a) \mathbin{\&} (c_2, b) = (c_1 \otimes_{\bullet} c_2, (a, b))$.

Parallel complexity of sorting. We have verified the sequential and parallel complexity of insertion sort and merge sort under the comparison cost model. As outlined above, we instantiate **calf** with the parallel cost monoid $\mathbb{N}^2$ in which the first component represents the sequential cost and the second component represents the parallel cost. The analysis is parameterized by a *comparable* type $A : \mathsf{tp}^+$ that is equipped with a comparison operation $\leq^b : \mathsf{tm}^+(A) \to \mathsf{tm}^+(A) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{bool}))$. Consequently, we may enforce the cost model by requiring the comparison operation $\leq^b$ to be uniformly unit cost, *i.e.* $\mathsf{isBounded}\,(\mathsf{bool}; x \leq^b y; (1,1))$ for all $x, y : \mathsf{tm}^+(A)$. We have mechanized the following asymptotically tight cost bounds:

Theorem 6.1 (`Examples.Sorting.Parallel.InsertionSort.sort≤sort/cost/closed`). *For all* $l : \mathsf{list}(A)$*, we have that* $\mathsf{isBounded}\,(\mathsf{list}(A); \mathit{isort}(l); (|l|^2, |l|^2))$.

Observe that sequential and parallel complexity coincide for insertion sort because there is no opportunity for parallelism in the algorithm. The standard merge sort algorithm enjoys a logarithmic speed up when the recursive calls are performed in parallel:

Theorem 6.2 (`Examples.Sorting.Parallel.MergeSort.sort≤sort/cost/closed`). *For all* $l : \mathsf{list}(A)$*, we have that* $\mathsf{isBounded}\,(\mathsf{list}(A); \mathit{msort}(l); (\lceil \log_2 |l| \rceil \cdot |l|, 2 \cdot |l| + \lceil \log_2 |l| \rceil))$.

To obtain a sublinear bound on parallel complexity, one must modify merge sort to also perform the merging step in parallel, an alteration that slightly increases the sequential complexity:

Theorem 6.3 (`Examples.Sorting.Parallel.MergeSortPar.sort≤sort/cost/closed`). *For all* $l : \mathsf{list}(A)$*, we have that* $\mathsf{isBounded}\left(\mathsf{list}(A); \mathit{msortPar}(l); \left(\lceil \log_2 (|l|+1) \rceil^2 \cdot |l|, \lceil \log_2 (|l|+1) \rceil^3\right)\right)$.

7 CONCLUSION

Three somewhat contradictory goals guide our type-theoretic approach to cost analysis:

(1) Expressiveness: the ability to codify the methods and results of informal algorithm analysis.
(2) Certification: programs and their cost bounds should bear their intended meaning.
(3) Composition: cost bounds should be composable.

Most extant cost analysis frameworks excel at two out of three of the above. Type systems defined by intrinsic cost-aware judgments [Hoffmann et al. 2012; Rajani et al. 2021; Wang et al. 2017] are certified by soundness theorems and admit composition by construction but lack expressiveness because cost bounds are often form-constrained and typing derivations cannot exploit complex

behavioral properties. The traditional method for cost accounting using the writer monad [Handley et al. 2019] provides an expressive and compositional framework for cost analysis, but this transparent instrumentation is not certified in the aforementioned sense because programs in the writer monad do not necessarily accumulate cost faithfully (see Section 1.8.3). Lastly, frameworks for cost analysis in the setting of program logics [Atkey 2010; Mével et al. 2019] may be transposed to type theory by working with a deep embedding of a programming language and its operational semantics inside type theory. Although this can be developed into an expressive and certified framework in the sense above, it is not compositional because one may speak about operational semantics only on closed terms and must quantify over closing instances for open terms.

In this paper we show that the three goals may be achieved simultaneously. First, the extensional fragment of **calf** constituents an ordinary dependent type theory, which furnished us a rich specification language to formulate two widely used algorithm analysis techniques and illustrate each through detailed case studies. Secondly, we see that **calf** programs account for cost faithfully because cost structure arises via an *abstract* computational effect; therefore it is not possible to define exotic programs that spuriously abandon accumulated costs or branch based on the cost component of an input. Lastly, the CBPV structure of cost effects induces a simple equational theory that enables compositional cost analysis. We conclude by suggesting two particularly pertinent directions for future investigations.

Automation. In practice the usability of any verification framework may be greatly improved by automating routine procedures or derivations. In the context of **calf** there are two immediate opportunities for automation. On the one hand, the recurrence extraction step in the method of recurrence relations (as defined in Section 3.2) may be automated in many cases by incorporating the mechanism of Kavvos et al. [2019]. On the other hand, proofs involving restricted forms of cost bounds may be automated either by recurrence solving (*e.g.* the Master theorem) or an automated system such as RaML [Hoffmann et al. 2012].

Full adequacy and partiality. It would be interesting to prove an adequacy result of the form presented in Kavvos et al. [2019] in which one defines a cost-aware embedding of a source language (equipped with an operational semantics) in the target language (in this case **calf**) and proves that the image of any source program is assigned the same cost as the cost of the source program induced by the operational semantics. In many cases the source language of interest admits general recursion; consequently one must arrange for **calf** to faithfully interpret non-terminating programs. For instance, we may equip **calf** with the partiality monad of Capretta or guarded recursion à la Birkedal et al. [2011]; Bizjak et al. [2016]. We believe that **calf** is expressive enough for us to prove an internal version of the adequacy result of Kavvos et al. [2019] as Paviotti et al. have done for PCF in guarded type theory.

8 DATA AVAILABILITY STATEMENT

calf has been implemented in the Agda proof assistant [Niu et al. 2021].

ACKNOWLEDGMENTS

We are grateful to Carlo Angiuli and Alex Kavvos for productive discussions on the topic of this research, and to Tristan Nguyen at AFOSR for his support.

This work was supported in part by AFOSR under grants MURI FA9550-15-1-0053, FA9550-19-1-0216, and FA9550-21-0009, in part by the National Science Foundation under award number CCF-1901381, and by AFRL through the NDSEG fellowship. 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, NSF, or AFRL.

REFERENCES

Umut A. Acar and Guy E. Blelloch. 2019. *Algorithms: Parallel and Sequential.* http:www.algorithms-book.com.

Thorsten Altenkirch and Ambrus Kaposi. 2016a. Normalisation by Evaluation for Dependent Types. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 52)*, Delia Kesner and Brigitte Pientka (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:16. https://doi.org/10.4230/LIPIcs.FSCD.2016.6

Thorsten Altenkirch and Ambrus Kaposi. 2016b. Type Theory in Type Theory Using Quotient Inductive Types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, St. Petersburg, FL, USA, 18–29. https://doi.org/10.1145/2837614.2837638

Mathieu Anel and André Joyal. 2021. Topo-logie. In *New Spaces in Mathematics: Formal and Conceptual Reflections*, Mathieu Anel and Gabriel Catren (Eds.). Vol. 1. Cambridge University Press, Chapter 4, 155–257. https://doi.org/10.1017/9781108854429.007

Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *Programming Languages and Systems*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 85–103.

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2011. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE Computer Society, Washington, DC, USA, 55–64. https://doi.org/10.1109/LICS.2011.16 arXiv:1208.3596 [cs.LO]

Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. In *Foundations of Software Science and Computation Structures: 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, Bart Jacobs and Christof Löding (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 20–35. https://doi.org/10.1007/978-3-662-49630-5_2 arXiv:1601.01586 [cs.LO]

Guy Blelloch and John Greiner. 1995. Parallelism in Sequential Functional Languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. Association for Computing Machinery, La Jolla, California, USA, 226–237. https://doi.org/10.1145/224164.224210

Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efficient Implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, Philadelphia, Pennsylvania, USA, 213–225. https://doi.org/10.1145/232627.232650

Ana Bove and Venanzio Capretta. 2005. Modelling general recursion in type theory. *Mathematical Structures in Computer Science* 15, 4 (2005), 671–708. https://doi.org/10.1017/S0960129505004822

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (Sept. 2013), 552–593. https://doi.org/10.1017/S095679681300018X

F. Burton. 1982. An Efficient Functional Implementation of FIFO Queues. *Inf. Process. Lett.* 14 (1982), 205–206.

Kevin Buzzard, Johan Commelin, and Patrick Massot. 2020. Formalising Perfectoid Spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Association for Computing Machinery, New Orleans, LA, USA, 299–312. https://doi.org/10.1145/3372885.3373830

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* 1, 2 (2005), 1–18.

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Robert L. Constable and Karl Crary. 2002. *Computational complexity and induction for partial computable functions in type theory.* Cambridge University Press, Cambridge, 164–181. https://doi.org/10.1017/9781316755983.009

Robert L. Constable and Daniel R. Zlatin. 1984. The Type Theory of PL/CV3. *ACM Transactions on Programming Languages and Systems* 6, 1 (Jan. 1984), 94–117. https://doi.org/10.1145/357233.357238

The Coq Development Team. 2016. The Coq Proof Assistant Reference Manual.

Thierry Coquand. 2019. Canonicity and normalization for dependent type theory. *Theoretical Computer Science* 777 (2019), 184–191. https://doi.org/10.1016/j.tcs.2019.01.015 arXiv:1810.09367 [cs.PL] In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

Karl Crary and Stephanie Weirich. 2000. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, Boston, MA, USA, 184–198. https://doi.org/10.1145/325694.325716

Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/1328438.1328457

Norman Danner, Daniel R. Licata, and Ramyaa. 2015. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). Association for Computing Machinery, 140–151. https://doi.org/10.1145/2784731.2784749

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018a. Parallel Complexity Analysis with Temporal Session Types. In *Proceedings of International Conference on Functional Programming (ICFP 2018)*, M. Flatt (Ed.). ACM, St. Louis, Missouri, USA, 91:1–91:30.

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018b. Work Analysis with Resource-Aware Session Types. In *Proceedings of 33rd Symposium on Logic in Computer Science (LICS 2018)*, A. Dawar and E. Grädel (Eds.). Oxford, UK, 305–314.

Ankush Das and Frank Pfenning. 2020. Rast: A Language for Resource-Aware Session Types. *CoRR* abs/2012.13129 (Dec. 2020). https://arxiv.org/abs/2012.13129 Submitted.

Rowan Davies and Frank Pfenning. 1999. A Modal Analysis of Staged Computation. *J. ACM* 48 (Sept. 1999). https://doi.org/10.1145/382780.382785

Manuel Eberl. 2015. The Akra-Bazzi theorem and the Master theorem. *Archive of Formal Proofs* (July 2015). https://isa-afp.org/entries/Akra_Bazzi.html, Formal proof development.

Manuel Eberl. 2017a. The Median-of-Medians Selection Algorithm. *Archive of Formal Proofs* (Dec. 2017). https://isa-afp.org/entries/Median_Of_Medians_Selection.html, Formal proof development.

Manuel Eberl. 2017b. The number of comparisons in QuickSort. *Archive of Formal Proofs* (March 2017). https://isa-afp.org/entries/Quick_Sort_Cost.html, Formal proof development.

Marcelo P. Fiore, Andrew M. Pitts, and S. C. Steenkamp. 2021. Quotients, inductive types, and quotient inductive types. (2021). arXiv:2101.02994 [cs.LO]

G. Gonthier. 2008. Formal Proof — The Four-Color Theorem. *Notices of the AMS* 55, 11 (2008). https://www.ams.org/notices/200811/tx081101382p.pdf

Daniel Gratzer and Jonathan Sterling. 2020. Syntactic categories for dependent type theory: sketching and adequacy. (2020). arXiv:2012.10783 [cs.LO]

John Greiner and Guy E. Blelloch. 1999. A Provably Time-Efficient Parallel Implementation of Full Speculation. *ACM Transactions on Programming Languages and Systems* 21, 2 (March 1999), 240–285. https://doi.org/10.1145/316686.316690

David Gries. 1987. *The Science of Programming* (1st ed.). Springer-Verlag, Berlin, Heidelberg.

Jesse Michael Han and Floris van Doorn. 2020. A Formal Proof of the Independence of the Continuum Hypothesis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Association for Computing Machinery, New Orleans, LA, USA, 353–366. https://doi.org/10.1145/3372885.3373826

Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2019. Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019). https://doi.org/10.1145/3371092

Robert Harper. 2018. **PFPL** Supplement: Types and Parallelism. (2018). https://www.cs.cmu.edu/~rwh/pfpl/supplements/par.pdf

Robert Harper, John C. Mitchell, and Eugenio Moggi. 1990. Higher-Order Modules and the Phase Distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, San Francisco, California, USA, 341–354. https://doi.org/10.1145/96709.96744

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 781–786.

Martin Hofmann. 2000. A Type System for Bounded Space and Functional In-Place Update–Extended Abstract. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*. Springer-Verlag, Berlin, Heidelberg, 165–179.

Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, New Orleans, Louisiana, USA, 185–197. https://doi.org/10.1145/604131.604148

Robert Hood and Robert Melville. 1981. Real-time queue operations in pure LISP. *Inform. Process. Lett.* 13, 2 (1981), 50–54. https://doi.org/10.1016/0020-0190(81)90030-2

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). Association for Computing Machinery, 223–236. https://doi.org/10.1145/1706299.1706327

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018),

e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning. In *POPL '15: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, Mumbai, India, 637–650. https://doi.org/10.1145/2676726.2676980

G. A. Kavvos. 2017a. Dual-Context Calculi for Modal Logic. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. arXiv:1602.04860 http://arxiv.org/abs/1602.04860

G. A. Kavvos. 2017b. *On the Semantics of Intensionality and Intensional Recursion*. Ph.D. Dissertation. arXiv:1712.09302

G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. 2019. Recurrence Extraction for Functional Programs through Call-by-Push-Value. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019). https://doi.org/10.1145/3371083

S. C. Kleene. 1943. Recursive predicates and quantifiers. *Trans. Amer. Math. Soc.* 53 (1943), 41–73. https://doi.org/10.2307/1990131

F. William Lawvere. 1963. *Functorial Semantics of Algebraic Theories*. Ph.D. Dissertation. Columbia University.

Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a Mechanized Metatheory of Standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, Nice, France, 173–184. https://doi.org/10.1145/1190216.1190245

Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2)*. Kluwer Academic Publishers, Norwell, MA, USA.

Paul Blain Levy. 2006. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation* 19 (2006), 377–414. https://doi.org/10.1007/s10990-006-0480-6

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.

Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gómez Londoño, Peter Lammich, Christian Sternagel, Simon Wimmer, and Bohua Zhan. 2021. *Functional Algorithms, Verified!* https://functional-algorithms-verified.org

Yue Niu and Robert Harper. 2020. Cost-Aware Type Theory. (2020). arXiv:2011.03660 [cs.PL]

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2021. `agda-calf`. https://doi.org/10.1145/3462303

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, Savannah, GA, USA, 1–2.

Chris Okasaki. 1998. *Purely Functional Data Structures*. Cambridge University Press, USA.

Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. 2015. A Model of PCF in Guarded Type Theory. *Electronic Notes in Theoretical Computer Science* 319, Supplement C (2015), 333–349. https://doi.org/10.1016/j.entcs.2015.12.020 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

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). https://doi.org/10.1145/3371126

Frank Pfenning. 2001. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, Washington, DC, USA, 221–. http://dl.acm.org/citation.cfm?id=871816.871845

G.D. Plotkin. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5, 3 (1977), 223–255. https://doi.org/10.1016/0304-3975(77)90044-5

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). https://doi.org/10.1145/3434308

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 arXiv:1706.07526 [math.CT]

Patrick Schultz and David I. Spivak. 2019. *Temporal Type Theory*. Progress in Computer Science and Applied Logic, Vol. 29. Birkhäuser Basel. https://doi.org/10.1007/978-3-030-00704-1 arXiv:1710.10258 [math.CT]

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2008. Space Profiling for Parallel Functional Programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, Victoria, BC, Canada, 253–264. https://doi.org/10.1145/1411204.1411240

Jonathan Sterling and Carlo Angiuli. 2021. Normalization for Cubical Type Theory. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–15. https://doi.org/10.1109/LICS52264.2021.9470719 arXiv:2101.11479 [cs.LO]

Jonathan Sterling and Robert Harper. 2021. Logical Relations as Types: Proof-Relevant Parametricity for Program Modules. *J. ACM* 68, 6 (Oct. 2021). https://doi.org/10.1145/3474834 arXiv:2010.08599 [cs.PL]

Aaron Stump. 2016. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool.

R. Tarjan. 1985. Amortized Computational Complexity. *Siam Journal on Algebraic and Discrete Methods* 6 (1985), 306–318.

Taichi Uemura. 2019. A General Framework for the Semantics of Type Theory. (2019). arXiv:1904.04097 [math.CT]

Sebastian Andreas Ullrich. 2016. *Simple Verification of Rust Programs via Functional Purification.* Master's thesis. IPD Snelting.

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017). https://doi.org/10.1145/3133903