Decalf: A Directed, Effectful Cost-Aware Logical Framework

HARRISON GRODIN, Carnegie Mellon University, USA
YUE NIU, Carnegie Mellon University, USA
JONATHAN STERLING, University of Cambridge, UK
ROBERT HARPER, Carnegie Mellon University, USA

We present **decalf**, a **d**irected, **e**ffectful **c**ost-**a**ware **l**ogical **f**ramework for studying quantitative aspects of functional programs with effects. Like **calf**, the language is based on a formal *phase distinction* between the *extension* and the *intension* of a program, its pure *behavior* as distinct from its *cost* measured by an effectful step-counting primitive. The type theory ensures that the behavior is unaffected by the cost accounting. Unlike **calf**, the present language takes account of *effects*, such as probabilistic choice and mutable state. This extension requires a reformulation of **calf**'s approach to cost accounting: rather than rely on a "separable" notion of cost, here *a cost bound is simply another program*. To make this formal, we equip every type with an intrinsic preorder, relaxing the precise cost accounting intrinsic to a program to a looser but nevertheless informative estimate. For example, the cost bound of a probabilistic program is itself a probabilistic program that specifies the distribution of costs. This approach serves as a streamlined alternative to the standard method of isolating a cost recurrence and readily extends to higher-order, effectful programs.

The development proceeds by first introducing the **decalf** type system, which is based on an intrinsic ordering among terms that restricts in the extensional phase to extensional equality, but in the intensional phase reflects an approximation of the cost of a program of interest. This formulation is then applied to a number of illustrative examples, including pure and effectful sorting algorithms, simple probabilistic programs, and higher-order functions. Finally, we justify **decalf** via a model in the topos of augmented simplicial sets.

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

Additional Key Words and Phrases: 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:
Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. 2024. Decalf: A Directed, Effectful Cost-Aware Logical Framework. *Proc. ACM Program. Lang.* 8, POPL, Article 10 (January 2024), 29 pages. https://doi.org/10.1145/3632852

1 INTRODUCTION

The **calf** language [Niu et al. 2022a] is a full-spectrum dependent type theory that consolidates the specification and verification of the (extensional) *behavior* and (intensional) *cost* of programs. For example, in **calf** it is possible to prove that insertion sort and merge sort are extensionally equal (*i.e.* have the same input/output behavior), and also that the former uses quadratically many

Authors' addresses: Harrison Grodin, Carnegie Mellon University, Computer Science Department, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, hgrodin@cs.cmu.edu; Yue Niu, Carnegie Mellon University, Computer Science Department, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, yuen@andrew.cmu.edu; Jonathan Sterling, University of Cambridge, Department of Computer Science and Technology, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK, js2878@cl.cam.ac.uk; Robert Harper, Carnegie Mellon University, Computer Science Department, 5000 Forbes Ave., Pittsburgh, PA, 15213, USA, rwh@cs.cmu.edu.

ACM 2475-1421/2024/1-ART10
https://doi.org/10.1145/3632852

comparisons for a given input list, whereas the latter uses poly-logarithmically many for the same input. Both programs are terms of the type theory, rather than encodings of programs internally to the type theory, and the stated properties are expressed as types.

It may seem, at first blush, that the stated properties cannot possibly be verified by typing—after all, if the sorting functions are *equal* as functions on lists, then how can they have *different* properties? Moreover, what does it mean, type-theoretically, for the two algorithms to require the stated number of comparisons? After all, it is not possible to determine *a posteriori* which piece of code is a comparison, much less determine how often it is executed. Furthermore, dependent type theory [Martin-Löf 1984] is given as an equational theory, so what would a comparison count even mean in such a setting?

The key to understanding how these questions are handled by **calf** lies in the combination of two developments in type theoretic semantics of programming languages:

(1) The view of cost as a computational effect, implemented equationally via Levy's *call-by-push-value* [Kavvos et al. 2019; Levy 2003].
(2) The reformulation of *phase distinctions* [Harper et al. 1990] in terms of open and closed modalities from topos theory, emanating from Sterling's Synthetic Tait Computability [Sterling 2021; Sterling and Harper 2021].

1.1 Polarity, Call-By-Push-Value, and Compositional Cost Analysis

In call-by-push-value, *values* of (positive) type are distinguished from *computations* of (negative) type.[1] Following the notation of Niu et al., we will typically write arbitrary positive types as A, B, C and negative types as X, Y, Z. Although not all models of call-by-push-value are of this form, it is instructive to think of negative types as being the algebras for a strong monad on the category of positive types and pure functions. Variables range over elements of positive types, which classify "passive" values such as booleans, numbers, tuples, and lists; computations inhabit negative types, which classify "active" computations, including the computation of values and, characteristically, functions mapping values to computations. Call-by-push-value can also accommodate type dependency more smoothly than either call-by-value or call-by-name [Ahman et al. 2016; Pédrot and Tabareau 2019; Vákár 2017].

The purpose of imposing polarity in **calf**'s dependent type theory via call-by-push-value is to give a compositional account of cost; in particular, **calf** instruments code with its cost by means of a write-only "step counting" effect $\mathsf{step}^c(e)$ that annotates computation e with an additional c cost, so determining the figure-of-merit for cost analysis. (In the case of sorting, the comparison operation is instrumented with an invocation of step counting on each usage.) The introduction of step counting as just described means that insertion sort and merge sort are *not equal* as functions on lists, exactly because they have different costs (but see Section 1.2). Moreover, their cost bounds can be characterized by saying, informally for the moment, that their steps counts on completion are related to the length of the input in the expected way.

1.2 A Phase Distinction Between Cost and Behavior

The **calf** type theory is therefore capable of expressing and verifying the usual sequential and parallel cost bounds on these two sorting algorithms, as well as for other algorithms as described in the paper of Niu et al. [2022a]. But what about their purely behavioral (extensional) equivalence? Here the second key idea comes into play, the introduction of a *phase distinction*, represented by a proposition (*i.e.* a type with at most one element) expressing that the *extensional phase* is in effect,

[1] Here borrowing terminology from polarization in proof theory to distinguish the two classes of types.

written $¶_{\text{ext}}$.[2] When $¶_{\text{ext}}$ is true/inhabited, *e.g.* by assumption in the context, then *the step counting computation is equal to the identity function*, and hence the two sorting algorithms are deemed *extensionally equal*.

1.2.1 Purely Extensional Types. It is best to understand the proposition $¶_{\text{ext}}$ as a *switch* that collapses all the cost information when activated/assumed. Then a type A is considered *purely extensional* when it "maintains the fiction" that $¶_{\text{ext}}$ is true: phrased more precisely, a purely extensional type is one for which the constant mapping $A \to (¶_{\text{ext}} \to A)$ is an isomorphism in the logical framework. Any type A can be restricted to its *extensional part*, namely the function space $\bigcirc A \coloneqq (¶_{\text{ext}} \to A)$. This *extensional modality* is also called an *open modality* in topos theory [Rijke et al. 2020].[3]

1.2.2 Purely Intensional Types. Dually, a type is considered *purely intensional* when it maintains the fiction that $¶_{\text{ext}}$ is *not* true; put in precise terms, A is purely intensional if and only if $\bigcirc A$ is a singleton, or (equivalently) if the projection map $¶_{\text{ext}} \times A \to ¶_{\text{ext}}$ is an isomorphism. In either case, the idea is that a purely intensional type A is one that becomes trivial (*i.e.* the unit type) within the extensional phase. This facility is used by **calf** to allow cost profiling information to be stored intensionally and then stripped away automatically in the extensional phase.[4]

1.2.3 Noninterference Between Intension and Extension. Under the phase distinction, intensional data has the *noninterference* property with respect to extensional data.

Proposition 1.1 (Noninterference). *Let A be a purely intensional type, and let B be a purely extensional type. Then any function $f : A \to B$ is* constant, i.e. *there is some $b_0 : B$ such that $f = \lambda\_.b_0$.*

Noninterference is crucial for realistic cost analysis: instrumenting programs with profiling data would not be conservative if the behavior of formalized programs could depend on that profiling data. Another reflection of noninterference is that we have an isomorphism $\bigcirc(A \to B) \cong (\bigcirc A \to \bigcirc B)$, which means that the behavior of a function is a function of the behaviors of its inputs.

1.3 Compositional Cost Analysis for Effectful Code

To motivate the contributions of this paper it is helpful to make more precise the informal discussion about cost and correctness in **calf**. Using again the examples of insertion sort and merge sort, the following facts about them can be verified in **calf**:

(1) Insertion sort is quadratic: $l : \mathsf{list}(\mathsf{nat}) \vdash \_ : \mathsf{isBounded}_{\mathsf{list}(\mathsf{nat})}(\mathit{isort}\ l, |l|^2)$.
(2) Merge sort is poly-logarithmic: $l : \mathsf{list}(\mathsf{nat}) \vdash \_ : \mathsf{isBounded}_{\mathsf{list}(\mathsf{nat})}(\mathit{msort}\ l, |l| \lg |l|)$.
(3) They are extensionally equal: $\bigcirc\,(\mathit{isort} = \mathit{msort})$.

Informally, the two sorting algorithms have the same input/output behavior, with insertion sort being notably less efficient than merge sort.

Here the relation $\mathsf{isBounded}_A(e, c)$, for computation $e : \mathsf{F}(A)$ and cost $c : \mathbb{C}$, is defined by

$$\mathsf{hasCost}_A(e, c) \coloneqq (a : A) \times (e = \mathsf{step}^c(\mathsf{ret}(a)))$$
$$\mathsf{isBounded}_A(e, c) \coloneqq (c' : \mathbb{C}) \times (c \leq_{\mathbb{C}} c') \times \mathsf{hasCost}_A(e, c')$$

The first defines the cost of a computation of type $\mathsf{F}(A)$ by an equation stating that it returns a value of type A with the specified cost, expressed as an equation between computations. The second states that the cost of a computation is at most the specified bound.[5]

[2]The logic of **calf** is *intuitionistic*, so an indeterminate proposition need not be either true or false in a given scope.
[3]Note that the extensional modality is nothing more than a function space: therefore, unlike in many other modal type theories, there is no need for sophisticated or nonstandard treatment of contexts (*e.g.* dual contexts, *etc.*).
[4]Although its inclusion poses no issues, we omit the complementary closed modality of **calf**, $\bullet A$, since it is not used here.
[5]Niu et al. include the extensional modality around the proof that $c \leq_{\mathbb{C}} c'$. For simplicity, we omit it here; the modality may be recovered by using cost model $\bigcirc\mathbb{C}$.

As long as the bounded expression is pure (free of effects other than the cost accounting itself), these definitions make good sense, and indeed these formulations have been validated empirically in the worst-case analysis of several algorithms [Niu et al. 2022a]. However, many algorithms rely on effects for their correctness and/or efficiency. For example, randomized algorithms require probabilistic sampling to ensure a good distribution on cost. Other algorithms rely on effects such as errors, nondeterminism, and mutable storage for correctness. As defined, **calf** does not account for any such behaviors. To be sure, it is entirely possible to extend **calf** with, say, a computation to probabilistically choose between two computations, and one can surely reason about the behavior of such programs, given sufficient libraries for reasoning about probabilities. The difficulty is with the definition of hasCost and isBounded: if e has effects other than cost profiling, then the definition of hasCost is not sensible because it neglects effects. For example, if e has effects on mutable storage, then the "final answer" must not only reflect the stepping effect, but also any "side effects" that it engenders as well. In the case of randomization the outcome of e, including its cost, is influenced by the probabilistic choice, which cannot, in general, be disregarded.

The definition of isBounded reflects a long-standing tendency in the literature to isolate a "mathematical" characterization of the cost of executing an algorithm by a function—typically recursively defined—that defines the number of steps taken as a function of (some abstraction of) its input. So, in the case of sorting, the functions $|l|^2$ and $|l| \lg |l|$ are mathematical functions that are used to specify the number of comparisons taken to sort a list l. Often the characterization is given by a *recurrence*, which is nothing other than a total recursive function of the cost parameter. For example, Niu et al. [2022a] analyze the cost of Euclid's algorithm in terms of the inverse to the Fibonacci sequence. While desirable when applicable, it cannot be expected in general that the cost can be so characterized. For example, one difficulty with the classical approach arises when higher-order functions are considered. In truth the cost of a computation that makes use of a function argument cannot be abstractly characterized in terms of "pure" costs, precisely because the cost of the algorithm depends on the full behavior of that function.

Effects, such as probabilistic sampling, introduce a similar difficulty, even in the case that the *outcome* is determinate, because the cost cannot be specified without reference to the source of randomness, its distribution. To put it pithily, a computation that incurs cost sampled from a binomial distribution is best characterized by the computation that implements the binomial distribution itself. The central question that drives the present work is this:

What better way to define the cost of an effectful program than by another effectful program?

After all, **calf** is a full-spectrum dependent type theory capable of formulating a very broad range of mathematical concepts—any appeal to an extrinsic formulation would defeat the very purpose of the present work, which is *to develop a synthetic account of cost analysis for effectful programs* that extends the established **calf** methodology for the special case of pure functional programs.

1.4 decalf: a Directed, Effectful Cost-Aware Logical Framework

The key to achieving these goals is to reformulate the **calf** type theory to consider *inequalities*, as well as equalities, on programs that, intuitively, express relaxations of the cost profiling information in a program. Reflexivity of the preorder means that it is always valid to say, in effect, that a piece of code "costs what it costs." In this sense a cost analysis can never "fail", but of course it is usually desirable to characterize the cost of a program more succinctly and informatively using, say, a closed form, whenever possible. Transitivity of the preorder means that bounds on bounds may be consolidated, facilitating modular reasoning. As regards the **calf** methodology, in **decalf** the role of the hasCost relation is replaced by equational reasoning, and the role of the isBounded

relation is replaced by inequational reasoning. Inequalities, like equalities, account for both cost and behavior (taking into account the extensional phase as appropriate).

Thus **decalf** may be described as a *directed* extension of **calf** to account for *effects*. The "directed" aspect is manifested by the inequational judgments, $a \leq a'$ for values of (positive) type A, and $e \leq e'$ for computations of (negative) type X.

- Extensionally, inequality is just equality: if $e \leq e'$, then $\bigcirc(e = e')$.
- Intensionally, inequality relaxes costs: if $c \leq c'$, then $\mathsf{step}^c(e) \leq \mathsf{step}^{c'}(e)$.

In the pure case (with profiling as the only effect), the original **calf** methodology is recovered by defining isBounded via inequality, aligning with the existing definition of hasCost:[6]

$$\mathsf{hasCost}_A(e, c) \coloneqq (a : A) \times (e = \mathsf{step}^c(\mathsf{ret}(a)))$$
$$\mathsf{isBounded}_A(e, c) \coloneqq (a : A) \times (e \leq \mathsf{step}^c(\mathsf{ret}(a)))$$

In the presence of other effects, though, this new definition of isBounded generalizes that of **calf**. For example, suppose e makes use of randomization; as long as all possible executions use at most c cost and return the same value, then we will have $\mathsf{isBounded}(e, c)$. Sometimes, we will wish to let the cost bound itself include effects. Then, we will use the inequality relation $e \leq e'$ more generally.

Furthermore, the inequality relation can compare programs at *all* computation types, whereas isBounded only compares at type $\mathsf{F}(A)$. To analyze a computation of type $A \to \mathsf{F}(B)$ in **calf**, one must quantify over values $a : A$ and then use isBounded at type B. In contrast, **decalf** allows computations of type $A \to \mathsf{F}(B)$ to be compared and analyzed directly.

Simultaneous analysis of cost and correctness. The inequality relation compares both the cost (for inequality) and the correctness (for equality) simultaneously. In general, this is essential: the *cost* of later computations may depend on the *behavioral/extensional part* of earlier computations. For example, suppose *reverse* is the usual list reverse algorithm with linear cost in the length of its input. Then, knowing only the cost of a computation $e : \mathsf{F}(\mathsf{list}(\mathsf{nat}))$ is not enough for determining the cost of $\mathsf{bind}(e; \mathit{reverse})$, which depends on (the length of) the result of e. Thus we see that the intension-extension phase distinction counterposes the *interference* of behavior with cost to the *noninterference* of cost with behavior (Section 1.2.3).

However, when the code exhibits *non*interference of behavior with cost, we may work with a notion of a *classical* bound of e by cost c that arises by postcomposition with $A \to 1$:

$$e; \mathsf{ret}(\star) \coloneqq \mathsf{bind}(e; \lambda\_.\mathsf{ret}(\star)) \leq \mathsf{step}^c(\mathsf{ret}(\star)).$$

This technique is useful for collapsing down multiple possible return values (Sections 3.2 and 3.3) that all have the same cost and aligns well with the traditional presentations of cost analysis in which cost and correctness are considered independently as a final result.

Synopsis. The remainder of this paper is organized as follows: In Section 2 we define the **decalf** type theory, which is used in the remainder of the paper. In Section 3 we formulate algorithms (some with effects) and derive their cost bounds. In Section 4 we justify **decalf** topos-theoretically. In Section 5 we summarize results, and in Section 6 we suggest directions for future work.

1.5 Related Work

As regards related work, the principal reference is Niu et al. [2022a] on which the present paper is based. Therein is provided a comprehensive comparison to related work on formalized cost analysis, all of which applies as well to the present setting. We call attention, in particular, to the central role of the (dependent) call-by-push-value formulation of type theory [Levy 2003; Pédrot and

[6] Notice that extensionally, both restrict to $(a : A) \times (e = \mathsf{ret}(a))$, only requiring that e return a value.

Tabareau 2019], and to the application of synthetic Tait computability [Sterling 2021] to integrate intensional and extensional aspects of program behavior. In subsequent work, Grodin and Harper [2023] perform amortized cost analyses coinductively by proving an equality between a realistic implementation of a data structure and a simpler specification implementation; we draw inspiration from this perspective, here considering inductive algorithms and generalizing to inequality.

Directed type theory and synthetic domain theory. The **decalf** type theory and its built-in inequality relation is closely related to the idea of *directed type theory* [Licata and Harper 2011; Riehl and Shulman 2017], which generalizes Martin-Löf's (bidirectional) identity types to account for directed identifications. Another important input to **decalf**'s design is synthetic domain theory [Hyland 1991; Phoa 1991], in which types are also equipped with an intrinsic preorder. Both of these inputs can be seen in our presheaf model of **decalf** (Section 4), which resembles *both* traditional models of higher category theory and directed type theory *and* (pre)sheaf models of synthetic domain theory [Fiore and Rosolini 1997, 2001]. Our method to isolate presheaves that behave like preorders via internal orthogonality comes from Fiore [1997], whose ideas we have combined with the modern accounts of internal orthogonal reflection of Christensen et al. [2020]; Rijke et al. [2020].

Integrating cost and behavior. As observed by Niu et al. in their work on **calf**, obtaining a cost bound on a program frequently depends on simultaneously verifying a *behavioral* invariant of the code/data structures involved. This requirement is fulfilled in the setting of **calf** because programs are nothing more than terms in a dependent type theory equipped with a cost effect; consequently one can use dependent type theory as a rich specification language for program behavior. In this aspect the **decalf** verification ethos remains unchanged from that of **calf**: although **decalf** lacks general dependent sums as a theory, we may still use the *judgmental* dependent types at the level of the *logical framework* (see Section 2.1) to encode the necessary program behaviors.

On the other hand, one may take a more stratified view on programs *vs.* logic, an approach exemplified by Radiček et al. [2017] in the context of *relational* cost analysis. To handle behavioral properties of programs, the authors introduce a type system U^{C} by integrating a version of HOL with a suitable equational theory on the *cost monad*. Aside from the well-known differences between verification in dependent type theory and program logics, the type system of Radiček et al. differs from **decalf** in several axes. First, U^{C} is designed around the verification of pure functional programs, whereas **decalf** is designed around the modular accommodation of different (possibly interacting) effects. Second, whereas U^{C} works with *additive* costs[7], **decalf** may be instantiated with any cost monoid. Lastly, Radiček et al. also propose a type system for *relational* cost analysis. In this paper we focus on unary program analysis to isolate the new ideas brought forth in **decalf**, but it should be possible to incorporate the techniques of relation cost analysis into a dependently-typed setting.

Cost analysis of higher-order functions. In **calf**, cost bounds are "global" in the sense that cost refinements such as hasCost and isBounded from Section 1.3 are only defined for the type of *free* computations $\mathsf{F}(A)$. In contrast, the presence of an intrinsic preorder structure on *every* type in **decalf** allows us to easily express the cost bounds of higher-order functions as *functions themselves* (see Section 3.3). In this sense **decalf** represents a departure from the standard dogma held by the cost analysis community in which cost as a notion is categorically segregated from programs. In our view this stratification often invites unnecessary duplication of structures, especially in the analysis of higher-order functions. This can be seen in *e.g.* the work of Rajani et al. on a type system called λ-amor for higher-order amortized analysis. Roughly, λ-amor deals with the pain points of

[7] Additivity was not precisely defined in Radiček et al. [2017]; we believe this refers to when a monoid is both commutative and satisfies the cancellation property.

traditional potential-based type systems by properly accounting for the affine structure of higher-order potential-carrying types. To accurately represent the cost structure of higher-order functions, λ-amor is also equipped with a language of *indices* resembling a simple functional language. Indices are a classic example of a "shadow language" in the sense that they simply abstract over the existing constructs of the associated programming language. By taking seriously the dictum "*cost bounds are programs*", we may dispense with such duplications of effort in **decalf** by means of *internal cost refinements* that are defined using ordinary type-theoretic constructions.

Algebraic effects. In Section 3, we extend **decalf** with various effect structures: nondeterminism, probabilistic choice, and a simple form of global state. Each effect is specified via an algebraic theory, drawing from foundational ideas on algebraic effects [Plotkin and Power 2002]. Note that here, we do not support handling of effects.

2 THE decalf TYPE THEORY

The **decalf** type theory is a dependent extension of Levy's call-by-push-value framework [Levy 2003] in which types are classified into one of two categories,

(1) *Value*, or *positive*, types are those whose elements are "pure data." This includes finite sums and products, inductive types such as the natural numbers or lists of a value type, and *suspensions* of computations.
(2) *Computation*, or *negative*, types are "active" and, in particular, may engender effects. The basic constructs are ret and bind, which incorporate values as computations, and sequence computations, respectively. Computation types include functions from positive to negative types, according to the intuition that functions may only be applied to values, and doing so engenders a computation. However, functions may be turned into values by suspension.

Crucially, *variables range over value types*, both in terms and in types themselves. That is, *families of values* and *families of types* are indexed by values, not computations, just as in the work of Pédrot and Tabareau [2019]. As in that work, the **decalf** type theory includes both positive and negative dependent functions and internalized equality and inequality types (more on the latter shortly). In contrast to *op. cit.*, **decalf** does not have dependent sums, but it does have positive products, non-dependent sums, and inductive types. In this regard, the **decalf** type theory is simpler than both **calf** [Niu et al. 2022a] or **∂cbpv** [Pédrot and Tabareau 2019].

The importance of the call-by-push-value–inspired formulation is that it is compatible with effects. In the case of **calf**, the sole effect under consideration is *cost profiling*, or *step-counting*, which is used to express the cost of a computation. For example, in the case of sorting, the comparison function is instrumented with a step operation that serves to count the number of comparisons, and permits the cost of sorting to be expressed in terms of this fundamental operation. In terms of semantics, the manifestation of the profiling effect is that computations types are interpreted as an algebra for the *writer monad*, which maintains state to accumulate the total cost of step operations that have been executed. Various forms of cost accounting (both sequential and parallel) are accounted for by parameterizing the type theory with respect to an ordered cost monoid. The present work seeks to extend **calf** to consider programs that engender effects besides profiling.

Phase distinction. As with **calf**, the **decalf** type theory includes a *phase distinction* between *extensional* and *intensional* aspects of programs. As in **calf** the extensional phase is governed by a proposition (a type with at most one element) named $\P_{\mathsf{ext}}$, whose assumption renders step counting inert. However, in contrast to **calf**, the expression of cost bounds in the presence of effects is much more delicate. In **calf** the outcome of a closed computation must be of the form $\mathsf{step}^c(\mathsf{ret}(v))$, which specifies both the cost, c, and the final value, v, of that program. However, in the presence of effects,

this simple characterization is not available. For example, the outcome of a probabilistic program is a *distribution* of such costs and behaviors derived from the distribution of its randomized inputs. Similarly, the outcome of an imperative computation must include accesses to the global state, because the final value may well refer to it. As in **calf**, the "open" modality $\bigcirc A \coloneqq (¶_{\text{ext}} \to A)$ is used to disregard the profiling effects to speak of pure behavior.

Program inequalities. To account for effects, the specification and verification of cost bounds in **decalf** is markedly different from that in **calf**. The most natural, and most general, way to specify the outcome of an effectful computation is by another effectful computation. Thus, just as a randomized algorithm uses coin flips to guide its behavior, so the cost specification correspondingly makes use of coin flips in its formulation. This might suggest an equational formulation of cost, until one considers that algorithm analysis is typically phrased in terms of *upper bounds* on cost. In **calf** that was handled by the preorder on the cost monoid, but to account for this in **decalf** the essential move is to introduce an *approximation ordering* among the programs of a type that permits these bounds to be relaxed. Thus, upper bounds on cost are expressed using the approximation ordering, written $e \le e'$, where e' captures some (weakening of) the cost of e.

In particular, in the case of purely functional programs (apart from step counting), the expression e' can, as in **calf**, be taken to have the form $\mathsf{step}^c(\mathsf{ret}(v))$. At the other extreme, the reflexivity of a preorder ensures that e' may always be taken to be e itself, specifying that "e costs whatever it costs"! This remark is not entirely frivolous. For one thing, there certainly are situations in which no better bound can be proved without significant simplifying assumptions. These situations arise especially when using higher-order functions whose cost and behavior may well be highly sensitive to the exact input, and not just some approximation thereof. Here approximation comes to the rescue, allowing one function to be approximated by another, and relying on the *monotonicity* of the approximation ordering to derive a useful upper bound. For example, let *map* be the usual higher-order list mapping function taking a function as an argument (Example 3.12). If $f \le f'$, then $\mathit{map}\ f \le \mathit{map}\ f'$, and the latter may well admit a meaningful upper bound—if, say, f' has constant cost, then, by transitivity of the approximation preorder, that constant may be used to derive a useful linear-time upper bound for that instance of list mapping.

2.1 Presentation of decalf in a Logical Framework

Following the formulation of Niu et al. [2022a], the **decalf** type theory is defined as a signature in an extensional, higher-order, dependently typed *logical framework* (LF). Specifically, a type theory is described by a list of constants in a version of extensional type theory with dependent product and sum types. The adequacy of the LF presentation of **decalf** with respect to the conventional presentations with contexts follows from conservativity of locally cartesian closed categories [Gratzer and Sterling 2020] over categories with representable maps [Uemura 2021, 2023].

The object theory **decalf** is specified with judgments declared as constants ending in **Jdg**, handling binding and scope of variables via the framework-level dependent product $(a : A) \to X(a)$.

2.2 Dependent Call-By-Push-Value Structure

First, we give a presentation of our core dependent call-by-push-value calculus:

$$\mathsf{tp}^+, \mathsf{tp}^\ominus : \mathbf{Jdg}$$
$$\mathsf{tm}^+ : \mathsf{tp}^+ \to \mathbf{Jdg}$$
$$\mathsf{tm}^\ominus : \mathsf{tp}^\ominus \to \mathbf{Jdg}$$
$$\mathsf{tm}^\ominus(X) \coloneqq \mathsf{tm}^+(\mathsf{U}(X))$$

$$\mathsf{U} : \mathsf{tp}^\ominus \to \mathsf{tp}^+$$
$$\mathsf{F} : \mathsf{tp}^+ \to \mathsf{tp}^\ominus$$
$$\mathsf{ret} : (A : \mathsf{tp}^+, a : \mathsf{tm}^+(A)) \to \mathsf{tm}^\ominus(\mathsf{F}(A))$$
$$\mathsf{bind} : \{A : \mathsf{tp}^+, X : \mathsf{tp}^\ominus\}\ \mathsf{tm}^\ominus(\mathsf{F}(A)) \to (\mathsf{tm}^+(A) \to \mathsf{tm}^\ominus(X)) \to \mathsf{tm}^\ominus(X)$$

Types in **decalf** are divided into two classes, the *positive* value types tp^+ and the *negative* computation types $\mathsf{tp}^\ominus$. Each type has a corresponding collection of terms, $\mathsf{tm}^+(A)$ and $\mathsf{tm}^\ominus(X)$. Following Niu et al. [2022a], we define computations as $\mathsf{tm}^\ominus(X) \coloneqq \mathsf{tm}^+(\mathsf{U}(X))$, leading to a less bureaucratic version of call-by-push-value in which thunk and force are identities.

The two levels are linked by a pair of modalities, $\mathsf{F}(A)$ and $\mathsf{U}(X)$, that, respectively, classify computations of a given value and reify such computations as values. The ret and bind constructs return values and sequence computations, as would be expected in a language with effects. Semantically, computation types are interpreted as algebras for a monad, which provides the structure required to consolidate effects as a computation proceeds. Equality of values, $a = a'$, and computations, $e = e'$, is the equality provided by the logical framework.

2.3 Type Structure

The **decalf** language includes both positive and negative dependent product types. As to the former, these are functions that map values to values, and hence must not have effects. Their applications constitute complex values in the sense of [Levy 2003, §3]. We make use of them for readability; for example, we freely use arithmetic operations to describe the amount of cost being incurred. The latter map values to computations, and hence may have effects, both profiling and any other effect that may be added to the language. The **decalf** type theory includes equality types with equality reflection, with the consequence that function equality is extensional (and hence undecidable in cases of interest).[8]

$$\times : \mathsf{tp}^+ \to \mathsf{tp}^+ \to \mathsf{tp}^+$$
$$(\mathsf{unpair}, \mathsf{pair}) : \{A, B\}\ \mathsf{tm}^+(A \times B) \cong (a : \mathsf{tm}^+(A)) \times \mathsf{tm}^+(B)$$

$$\mathsf{eq} : (A : \mathsf{tp}^+) \to \mathsf{tm}^+(A) \to \mathsf{tm}^+(A) \to \mathsf{tp}^+$$
$$(\mathsf{ref}, \mathsf{self}) : \{A\}\ (a, a' : \mathsf{tm}^+(A)) \to \mathsf{tm}^+(\mathsf{eq}_A(a, a')) \cong (a = a')$$

$$\Pi^+ : (A : \mathsf{tp}^+, B : \mathsf{tm}^+(A) \to \mathsf{tp}^+) \to \mathsf{tp}^+$$
$$(\mathsf{ap}^+, \mathsf{lam}^+) : \{A, B\}\ \mathsf{tm}^+(\Pi^+(A; B)) \cong (a : \mathsf{tm}^+(A)) \to \mathsf{tm}^+(B(a))$$

$$\Pi : (A : \mathsf{tp}^+, X : \mathsf{tm}^+(A) \to \mathsf{tp}^\ominus) \to \mathsf{tp}^\ominus$$
$$(\mathsf{ap}, \mathsf{lam}) : \{A, X\}\ \mathsf{tm}^\ominus(\Pi(A; X)) \cong (a : \mathsf{tm}^+(A)) \to \mathsf{tm}^\ominus(X(a))$$

The language is also equipped with standard positive types, including (non-dependent) sum types and inductive types (such as natural numbers and lists), whose definitions are included in the appendix of the extended version [Grodin et al. 2023]. Importantly, the elimination forms for these types take as motives families of *judgments*, not just types. The reason for this is to support so-called "large eliminations," families of types indexed by sums and inductive types.

2.4 Reasoning About Extensional Properties Using $¶_{\mathsf{ext}}$

In general, programs, equations, and inequalities in **decalf** take cost structure into account. To consider only cost-ignoring behavioral properties, we study programs in the fragment of **decalf** under the extensional phase, $¶_{\mathsf{ext}}$:

$$¶_{\mathsf{ext}} : \mathbf{Jdg}$$
$$¶_{\mathsf{ext}}/\mathsf{uni} : \{u, v : ¶_{\mathsf{ext}}\}\ u = v$$
$$\bigcirc(\mathcal{J}) \coloneqq ¶_{\mathsf{ext}} \to \mathcal{J}$$

[8] Of course, the famous conservativity result of Hofmann [1995] shows that **decalf** can nonetheless be adequately formalized in a decidable intensional type theory with enough axioms; this principle is used already in the Agda mechanization of **calf**, where Agda is extended with **calf**-specific primitives and axioms as postulates.

Here, $\bigcirc(-)$ is the extensional modality, governing *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 **decalf** as the type $\bigcirc(e = e')$.

2.5 Preorder Structure on Types

The approximation preorder on **decalf** values and computations is induced by these principles:

(1) All functions are (automatically) monotone.
(2) Functions are compared pointwise.
(3) Under the extensional phase, $a \leq a'$ implies $a = a'$.

The extensionality requirement expresses that the approximation ordering is solely to do with cost: when cost effects are suppressed, the preorder is just equality, and thus has no effect on the behavior of the program. We render these conditions formally in the logical framework as follows:

$$\leq : \{A : \mathsf{tp}^+\}\ \mathsf{tm}^+(A) \to \mathsf{tm}^+(A) \to \mathbf{Jdg}$$
$$\leq_{\mathsf{isPreorder}} : (A : \mathsf{tp}^+) \to \mathsf{isPreorder}(A, \leq)$$
$$\leq_{\mathsf{mono}} : \{A, B, a, a'\}\ (f : \mathsf{tm}^+(A) \to \mathsf{tm}^+(B)) \to a \leq a' \to f(a) \leq f(a')$$
$$\leq_{\mathsf{pi}} : \{A, B, f, f'\}\ ((a : \mathsf{tm}^+(A)) \to f(a) \leq f'(a)) \to \mathsf{lam}(f) \leq \mathsf{lam}(f')$$
$$\leq_{\mathsf{ext}} : \P_{\mathsf{ext}} \to a \leq a' \to a = a'$$

By the definition of $\mathsf{tm}^\ominus(X)$, computations of type X are compared at value type $\mathsf{U}(X)$. We may also internalize this judgmental structure:

$$\mathsf{leq} : (A : \mathsf{tp}^+) \to \mathsf{tm}^+(A) \to \mathsf{tm}^+(A) \to \mathsf{tp}^+$$
$$(\mathsf{lref}, \mathsf{lself}) : \{A\}\ (a, a' : \mathsf{tm}^+(A)) \to \mathsf{tm}^+(\mathsf{leq}_A(a, a')) \cong (a \leq a')$$

2.6 Cost Monoid: Cost Structure of Programs

Cost-aware programs carry quantitative information through elements of the cost monoid $\mathbb{C}$, which is a positive type in **decalf**. Because different algorithms and cost models require different notions of cost, we parameterize **decalf** by a *purely intensional monoid* $(\mathbb{C}, +, 0)$ in the sense of Section 1.2.2; in other words $\mathbb{C}$ becomes a singleton in the extensional phase.[9]

$$\mathbb{C} : \mathsf{tp}^+$$
$$\mathbb{C}/\P_{\mathsf{ext}} : \bigcirc(\mathsf{tm}^+(\mathbb{C}) \cong \mathbf{1})$$
$$0 : \mathsf{tm}^+(\mathbb{C})$$
$$+ : \mathsf{tm}^+(\mathbb{C}) \to \mathsf{tm}^+(\mathbb{C}) \to \mathsf{tm}^+(\mathbb{C})$$
$$\mathsf{costMon} : \mathsf{isMonoid}(\mathbb{C}, 0, +)$$

Since every type is equipped with an intrinsic preorder, this automatically makes $(\mathbb{C}, +, 0, \leq_{\mathbb{C}})$ a preordered monoid.

2.7 Cost as an Effect in decalf

As in **calf**, costs in **decalf** are formulated in terms of a profiling computation $\mathsf{step}^c_X(e)$ that is parameterized by a computation type X and an element of the cost type c. The meaning of $\mathsf{step}^c_X(e)$ is to charge c units of cost and continue as e; consequently, we require that step is coherent with the monoid structure on $\mathbb{C}$.

$$\mathsf{step} : \{X : \mathsf{tp}^\ominus\}\ \mathsf{tm}^+(\mathbb{C}) \to \mathsf{tm}^\ominus(X) \to \mathsf{tm}^\ominus(X)$$
$$\mathsf{step}_0 : \{X, e\}\ \mathsf{step}^0(e) = e$$
$$\mathsf{step}_+ : \{X, c_1, c_2, e\}\ \mathsf{step}^{c_1}(\mathsf{step}^{c_2}(e)) = \mathsf{step}^{c_1 + c_2}(e)$$

[9] Observe that this implies that we have $\mathsf{step}^c(e) = e$ for every cost $c : \mathbb{C}$ in the extensional phase.

$$
\begin{aligned}
&\mathit{double} : \mathsf{nat} \to \mathsf{F}(\mathsf{nat}) \\
&\mathit{double}\ \mathsf{zero} = \mathsf{ret}(\mathsf{zero}) \\
&\mathit{double}\ (\mathsf{suc}(n)) = \mathsf{step}^1(\mathsf{bind}\ n' \leftarrow \mathit{double}\ n\ \mathsf{in}\ \mathsf{ret}(\mathsf{suc}(\mathsf{suc}(n'))))
\end{aligned}
$$

Fig. 1. Recursive implementation of the doubling function on natural numbers, instrumented with one cost at each recursive call.

In addition, we require equations governing the interaction of step with other computations, as is standard in call-by-push-value.

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

As a consequence of monotonicity, if $c \leq c'$ holds, then we have $\mathsf{step}^c_X(e) \leq \mathsf{step}^{c'}_X(e)$ for any computation $e : X$. This is the means by which cost bounds are relaxed in an analysis.

3 VERIFICATION EXAMPLES

Equipped with equality and inequality of programs, we now provide examples of how a cost analysis may be performed. In place of a cost bound, we simply use another program. The purpose of cost analysis, then, will be to condense the details of a complex program.

In the forthcoming examples, we will instantiate the cost model to $(\omega, \leq_\omega, +, 0)$, the natural numbers with the usual ordering and additive monoid structure. Note that this object is not the inductive type of natural numbers, whose preorder is discrete. We note that ω is a purely intensional type and provide a description of ω as a *quotient inductive type* [Kaposi et al. 2019] in Section 4.5.1.

3.1 Pure Algorithms

First, we discuss pure algorithms in which cost is the only available effect, presenting the work of Niu et al. [2022a] in **decalf**. In this case, a cost bound will typically look like a closed form for the program at hand.

Example 3.1 (Doubling function). Consider the function that recursively doubles a natural number, given in Fig. 1, where we annotate with one unit of cost per recursive call. Its behavior can be concisely specified via a non-recursive closed form:

$$\mathit{double}_{\mathsf{bound}} \coloneqq \lambda n.\, \mathsf{step}^n(\mathsf{ret}(2n))$$

Here, $2n$ is a complex value used to specify the return value. By induction, we may prove that *double* is equal to the closed form $\mathit{double}_{\mathsf{bound}}$. This fact constitutes a proof of both the cost and correctness of *double*. As a corollary, we may isolate a correctness-only proof using the extensional phase: $\bigcirc(\mathit{double} = \lambda n.\, \mathsf{ret}(2n))$. ⌟

Although this approach is new, the same reasoning is valid in **calf**. However, not every algorithm has a simple closed form; sometimes, we may only wish to give an upper bound.

Example 3.2 (List insert). Consider the implementation of *insert*, a subroutine of insertion sort, in Fig. 2. Here, we count the number of comparison operations performed. The cost incurred by the computation *insert* x l depends on the particular elements of the list l in relation to the value x. To characterize its cost precisely, we could of course take *insert* as its own cost bound, since insert is equal to itself by reflexivity. However, this bound provides more detail than a client may wish for. Rather than characterize this cost precisely, then, it is common to give only an upper bound.

In the worst case, *insert* x l incurs $|l|$ cost, when x is placed at the end of l. Thus, we may define $\mathit{insert}_{\mathsf{bound}} \coloneqq \lambda x.\, \lambda l.\, \mathsf{step}^{|l|}(\mathsf{ret}(\mathit{insert}_{\mathsf{spec}}\ x\ l))$. Here, $\mathit{insert}_{\mathsf{spec}}$ is a complex value specifying the

$$
\begin{array}{l}
\mathit{insert} : \mathsf{nat} \to \mathsf{list}(\mathsf{nat}) \to \mathsf{F}(\mathsf{list}(\mathsf{nat})) \\
\mathit{insert}\ x\ [] = \mathsf{ret}(x :: []) \\
\mathit{insert}\ x\ (y :: ys) = \\
\quad \mathsf{bind}\ b \leftarrow \mathsf{step}^1(x \leq^?\ y)\ \mathsf{in} \\
\quad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{ret}(x :: y :: ys)\ \mathsf{else}\ \mathsf{bind}\ ys' \leftarrow \mathit{insert}\ x\ ys\ \mathsf{in}\ \mathsf{ret}(x :: ys')
\end{array}
$$

Fig. 2. Recursive implementation of the list insertion, the auxiliary function of insertion sort, instrumented with one cost per element comparison.

$$
\begin{array}{l}
\mathit{isort} : \mathsf{list}(\mathsf{nat}) \to \mathsf{F}(\mathsf{list}(\mathsf{nat})) \\
\mathit{isort}\ [] = \mathsf{ret}([]) \\
\mathit{isort}\ (x :: xs) = \mathsf{bind}\ xs' \leftarrow \mathit{isort}\ xs\ \mathsf{in}\ \mathit{insert}\ x\ xs'
\end{array}
$$

Fig. 3. Insertion sort algorithm, using auxiliary *insert* function from Fig. 2. Cost is not directly instrumented here, but a call to *isort* counts comparisons based on the implementation of *insert*.

intended behavior of the *insert* computation. Using program inequality of **decalf**, we prove by induction that *insert* is bounded by this closed form in the sense that $\mathit{insert} \leq \mathit{insert}_{\mathrm{bound}}$. As in Example 3.1, this fact constitutes a proof of both the cost and correctness of *insert*. In terms of cost, it shows that $\mathit{insert}\ x\ l$ incurs at most $|l|$ cost. The inequality of programs only has an impact on cost; thus, this proof also guarantees that $\mathit{insert}\ x\ l$ returns $\mathit{insert}_{\mathrm{spec}}\ x\ l$. Since inequality is extensionally equality, we achieve the following extensional correctness guarantee as a corollary of the inequality above: $\bigcirc(\mathit{insert} = \lambda x.\, \lambda l.\, \mathsf{ret}(\mathit{insert}_{\mathrm{spec}}\ x\ l))$. ⌟

Remark 3.3. The implementation of $\mathit{insert}_{\mathrm{spec}}$ here corresponds to a component of the cost bound proofs of Niu et al. [2022a]. There, the specification is implemented inline in the cost bound proof as the value that the computation proved to return. Here, we reorganize the data, giving the specification implementation first and using it to prove a program bound.

Example 3.4 (Insertion sort). In Fig. 3, we show the implementation of the insertion sort algorithm, using auxiliary function *insert* from Fig. 2. As in Example 3.2, we may define a bounding program:

$$\mathit{isort}_{\mathrm{bound}} := \lambda l.\, \mathsf{step}^{|l|^2}(\mathsf{ret}(\mathit{sort}_{\mathrm{spec}}\ l))$$

Then, we may prove by induction that $\mathit{isort} \leq \mathit{isort}_{\mathrm{bound}}$.

Although it is less common than proving an upper bound, we may also show a lower bound of a computation. Since $\mathit{insert}\ x\ l$ costs at least 1 on a non-empty list l and *isort* is length-preserving, we can show that $\mathit{isort}\ l$ incurs at least $|l| - 1$ cost:

$$\lambda l.\, \mathsf{step}^{|l|-1}(\mathsf{ret}(\mathit{sort}_{\mathrm{spec}}\ l)) \leq \mathit{isort}$$ ⌟

Adapting the work by Niu et al. [2022a], we may also define the merge sort algorithm, *msort*, and prove that it is bounded by cost $|l| \lg |l|$:

$$\mathit{msort} \leq \lambda l.\, \mathsf{step}^{|l| \lg |l|}(\mathsf{ret}(\mathit{sort}_{\mathrm{spec}}\ l)).$$

Using the fact that program inequality is extensionally equality, though, we may recover the proof that these two sorting algorithms are extensionally equal.

Theorem 3.5. *In the extensional phase, we have that isort = msort.*

Proof. Extensionally, the inequalities in the cost bounds are equalities and the cost operation is trivialized. So, $\mathit{isort} = \mathit{isort}_{\mathrm{bound}} = \lambda l.\, \mathsf{ret}(\mathit{sort}_{\mathrm{spec}}\ l) = \mathit{msort}_{\mathrm{bound}} = \mathit{msort}$. □

$$\begin{aligned}
&\mathsf{branch} : \{X : \mathsf{tp}^{\ominus}\}\ \mathsf{tm}^{\ominus}(X) \to \mathsf{tm}^{\ominus}(X) \to \mathsf{tm}^{\ominus}(X)\\
&\mathsf{fail} : \{X : \mathsf{tp}^{\ominus}\}\ \mathsf{tm}^{\ominus}(X)\\
&\mathsf{branch/id}^{\mathsf{l}} : \{X, e\}\ \mathsf{branch}(\mathsf{fail},\ e) = e\\
&\mathsf{branch/id}^{\mathsf{r}} : \{X, e\}\ \mathsf{branch}(e,\ \mathsf{fail}) = e\\
&\mathsf{branch/assoc} : \{X, e_0, e_1, e_2\}\ \mathsf{branch}(\mathsf{branch}(e_0,\ e_1),\ e_2) = \mathsf{branch}(e_0,\ \mathsf{branch}(e_1,\ e_2))\\
&\mathsf{branch/comm} : \{X, e_0, e_1\}\ \mathsf{branch}(e_0,\ e_1) = \mathsf{branch}(e_1,\ e_0)\\
&\mathsf{branch/idem} : \{X, e\}\ \mathsf{branch}(e,\ e) = e\\
&\mathsf{branch/step} : \{X, c, e_0, e_1\}\ \mathsf{step}^{c}_{X}(\mathsf{branch}(e_0,\ e_1)) = \mathsf{branch}(\mathsf{step}^{c}_{X}(e_0),\ \mathsf{step}^{c}_{X}(e_1))\\
&\mathsf{fail/step} : \{X, c\}\ \mathsf{step}^{c}_{X}(\mathsf{fail}) = \mathsf{fail}
\end{aligned}$$

Fig. 4. Specification of the branch and fail primitives for finitary nondeterministic branching, which form a semilattice. The laws for interactions with the step primitive are included. Note that fail is the nullary correspondent to branch, so just as branch/step pushes cost into all two branches, fail/step pushes cost into all zero branches. The laws for interactions with computation type primitives are omitted for brevity, as they are analogous to those described in Section 2.7.

$$\begin{aligned}
&\mathit{choose} : \mathsf{list}(\mathsf{nat}) \to \mathsf{F}(\mathsf{nat} \times \mathsf{list}(\mathsf{nat}))\\
&\mathit{choose}\ [] = \mathsf{fail}\\
&\mathit{choose}\ (x :: xs) =\\
&\quad \mathsf{branch}(\mathsf{bind}\ (\mathit{pivot},\ l) \leftarrow \mathit{choose}\ xs\ \mathsf{in}\ \mathsf{ret}(\mathit{pivot},\ x :: l),\ \mathsf{ret}(x,\ xs))
\end{aligned}$$

$$\begin{aligned}
&\mathit{partition} : \mathsf{nat} \to \mathsf{list}(\mathsf{nat}) \to \mathsf{F}(\mathsf{list}(\mathsf{nat}) \times \mathsf{list}(\mathsf{nat}))\\
&\mathit{partition}\ \mathit{pivot}\ [] = \mathsf{ret}([],\ [])\\
&\mathit{partition}\ \mathit{pivot}\ (x :: xs) =\\
&\quad \mathsf{bind}\ (xs_1,\ xs_2) \leftarrow \mathit{partition}\ \mathit{pivot}\ xs\ \mathsf{in}\\
&\quad \mathsf{bind}\ b \leftarrow \mathsf{step}^{1}(x \leq^{?} \mathit{pivot})\ \mathsf{in}\\
&\quad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{ret}(x :: xs_1,\ xs_2)\ \mathsf{else}\ \mathsf{ret}(xs_1,\ x :: xs_2)
\end{aligned}$$

$$\begin{aligned}
&\mathit{qsort} : \mathsf{list}(\mathsf{nat}) \to \mathsf{F}(\mathsf{list}(\mathsf{nat}))\\
&\mathit{qsort}\ [] = \mathsf{ret}([])\\
&\mathit{qsort}\ (x :: xs) =\\
&\quad \mathsf{bind}\ (\mathit{pivot},\ l) \leftarrow \mathit{choose}\ (x :: xs)\ \mathsf{in}\\
&\quad \mathsf{bind}\ (l_1,\ l_2) \leftarrow \mathit{partition}\ \mathit{pivot}\ l\ \mathsf{in}\\
&\quad \mathsf{bind}\ l'_1 \leftarrow \mathit{qsort}\ l_1\ \mathsf{in}\\
&\quad \mathsf{bind}\ l'_2 \leftarrow \mathit{qsort}\ l_2\ \mathsf{in}\\
&\quad \mathsf{ret}(l'_1 +\!\!+ (x :: []) +\!\!+ l'_2)
\end{aligned}$$

Fig. 5. Quicksort algorithm [Hoare 1961, 1962], where the *choose* auxiliary function chooses a pivot nondeterministically. As in Figs. 2 and 3, the cost instrumentation tracks one unit of cost per comparison.

3.2 Effectful Algorithms

Treating cost bounds as program inequalities, we can extend **decalf** with various computational effects and prove bounds on effectful programs.

3.2.1 Nondeterminism. First, we consider the nondeterminism effect, specified in Fig. 4. Here, we specify finitary nondeterministic branching as a semilattice structure. The identity element for a binary nondeterministic branch is called fail, since nullary nondeterminism is akin to failure.

$$
\begin{array}{l}
\mathit{lookup} : \mathsf{list}(A) \to \mathsf{nat} \to \mathsf{F}(A) \\
\mathit{lookup}\ [\,]\ i = \mathsf{fail} \\
\mathit{lookup}\ (x :: xs)\ \mathsf{zero} = \mathsf{ret}(x) \\
\mathit{lookup}\ (x :: xs)\ (\mathsf{suc}(i)) = \mathsf{step}^1(\mathit{lookup}\ xs\ i)
\end{array}
$$

Fig. 6. The usual implementation of list index lookup, instrumented with one cost per recursive call. If the desired index is out of bounds, the computation errors via the fail effect.

Example 3.6 (Quicksort). In Fig. 5, we define a variant of the quicksort algorithm [Hoare 1961, 1962] in which the pivot is chosen nondeterministically. The number of comparisons computed by $\mathit{qsort}\ l$ depends on which element is chosen as a pivot; in the worst case, it can compute $|l|^2$ comparisons. We may prove this by induction:

$$\mathit{qsort} \le \lambda l.\, \mathsf{step}^{|l|^2}(\mathsf{ret}(\mathit{sort}_{\mathrm{spec}}\ l)).$$

The fact that $\mathit{qsort}\ l$ is nondeterministic is not reflected in this bound: regardless of the chosen pivot, it always incurs at most $|l|^2$ cost and returns $\mathit{sort}_{\mathrm{spec}}\ l$. In other words, the use of nondeterminism was *benign*: extensionally, it is invisible, rendering the program effect-free. ⌟

In the following examples, we consider situations where the effect is *not* benign and thus appears in the bounding program.

Example 3.7 (List lookup). In Fig. 6, we define a function *lookup* that finds an element of a list at a given index. Here, the cost model is that one cost should be incurred per recursive call. In case the list is shorter than the desired index, the program performs the fail effect, terminating the program. This effect is not benign: even extensionally, the impact of fail is visible. Therefore, any bound for this program must involve the fail effect, as well. We can define an exact bound for *lookup*:

$$\mathit{lookup}_{\mathrm{bound}} \coloneqq \lambda l.\, \lambda i.\, \mathsf{if}\ i <^? |l|\ \mathsf{then}\ \mathsf{step}^i(\mathsf{ret}(\mathit{lookup}_{\mathrm{spec}}\ l\ i))\ \mathsf{else}\ \mathsf{fail}$$

We may then prove by induction that $\mathit{lookup} \le \mathit{lookup}_{\mathrm{bound}}$. Extensionally, this says that

$$\mathit{lookup} = \lambda l.\, \lambda i.\, \mathsf{if}\ i <^? |l|\ \mathsf{then}\ \mathsf{ret}(\mathit{lookup}_{\mathrm{spec}}\ l\ i)\ \mathsf{else}\ \mathsf{fail},$$

which is the desired behavioral specification. ⌟

Example 3.8 (Pervasive nondeterminism). Sometimes, nondeterminism has an impact on the output of a program. For example, consider the following program:

$$e \coloneqq \mathsf{branch}(\mathsf{step}^3(\mathsf{ret}(\mathsf{true})),\ \mathsf{step}^{12}(\mathsf{ret}(\mathsf{false})))$$

Using the monotonicity of inequality and the laws for branch, it is possible to bound e as follows:

$$e \le \mathsf{step}^{12}(\mathsf{branch}(\mathsf{ret}(\mathsf{true}),\ \mathsf{ret}(\mathsf{false})))$$

However, for programs with more branching, specifying the correctness alongside the cost may add undesired noise. Thus, one may wish to bound the computation $e; \mathsf{ret}(\star)$ instead:

$$e; \mathsf{ret}(\star) \le \mathsf{step}^{12}(\mathsf{ret}(\star)).$$ ⌟

3.2.2 Probabilistic Choice. Similar to nondeterminism, we consider the (finitely-supported) probabilistic choice effect, where the nondeterminism is weighted by a rational number between 0 and 1. We specify the signature in Fig. 7 as the axioms for a convex space, where $\mathsf{flip}_p(e_0,\ e_1)$ takes the p-weighted combination of computations e_0 and e_1: with probability p compute e_1, and with probability $1 - p$ compute e_0.

$$\mathsf{flip} : \{X : \mathsf{tp}^{\ominus}\}\ \mathbb{Q}_{[0,1]} \to \mathsf{tm}^{\ominus}(X) \to \mathsf{tm}^{\ominus}(X) \to \mathsf{tm}^{\ominus}(X)$$
$$\mathsf{flip}/0 : \{X, e_0, e_1\}\ \mathsf{flip}_0(e_0,\ e_1) = e_0$$
$$\mathsf{flip}/1 : \{X, e_0, e_1\}\ \mathsf{flip}_1(e_0,\ e_1) = e_1$$
$$\mathsf{flip}/\mathsf{assoc} : \{X, p, q, r, e_0, e_1, e_2\}\ (p = 1 - ((1 - pq)(1 - r)))$$
$$\to \mathsf{flip}_{pq}(\mathsf{flip}_r(e_0,\ e_1),\ e_2) = \mathsf{flip}_p(e_0,\ \mathsf{flip}_q(e_1,\ e_2))$$
$$\mathsf{flip}/\mathsf{comm} : \{X, p, e_0, e_1\}\ \mathsf{flip}_p(e_0,\ e_1) = \mathsf{flip}_{1-p}(e_1,\ e_0)$$
$$\mathsf{flip}/\mathsf{idem} : \{X, p, e\}\ \mathsf{flip}_p(e,\ e) = e$$
$$\mathsf{flip}/\mathsf{step} : \{X, c, p, e_0, e_1\}\ \mathsf{step}^c_X(\mathsf{flip}_p(e_0,\ e_1)) = \mathsf{flip}_p(\mathsf{step}^c_X(e_0),\ \mathsf{step}^c_X(e_1))$$

Fig. 7. Specification of the flip primitive for finitary probabilistic choice, which forms a convex space. The law for interaction with the step primitive is included. Like in Fig. 4, the laws for interactions with computation type primitives are omitted for brevity, as they are analogous to those described in Section 2.7.

$$\mathit{sublist} : \mathsf{list}(\mathsf{nat}) \to \mathsf{F}(\mathsf{list}(\mathsf{nat}))$$
$$\mathit{sublist}\ [\,] = \mathsf{ret}([\,])$$
$$\mathit{sublist}\ (x :: xs) =$$
$$\quad \mathsf{bind}\ xs' \leftarrow \mathit{sublist}\ xs\ \mathsf{in}$$
$$\quad \mathsf{flip}_{1/2}(\mathsf{ret}(xs'),\ \mathsf{step}^1(\mathsf{ret}(x :: xs')))$$

Fig. 8. Implementation of *sublist*, an algorithm to compute a random sublist of an input list, where one unit of cost is incurred for each ::-node in the output list.

$$\mathit{bernoulli} : \mathsf{F}(1)$$
$$\mathit{bernoulli} = \mathsf{flip}_{1/2}(\mathsf{ret}(\star),\ \mathsf{step}^1(\mathsf{ret}(\star)))$$

$$\mathit{binomial} : \mathsf{nat} \to \mathsf{F}(1)$$
$$\mathit{binomial}\ \mathsf{zero} = \mathsf{ret}(\star)$$
$$\mathit{binomial}\ (\mathsf{suc}(n)) = \mathsf{bind}\ \star \leftarrow \mathit{bernoulli}\ \mathsf{in}\ \mathit{binomial}\ n$$

Fig. 9. Implementation of the Bernoulli and binomial cost distributions with $p = 1/2$.

The worst-case analysis of a probabilistic algorithm is analogous to the worst-case analysis of its nondeterministic counterpart. For example, one could define a randomized variant of *qsort* and show that its worst-case behavior is quadratic. Some algorithms, though, can be given tighter bounds based on their probabilistic weights.

Example 3.9 (Random sublist). In Fig. 8, we describe *sublist*, an algorithm for selecting a random sublist of an input list. We keep an element with probability ½ and count the number of ::-nodes as our cost model. It is not obvious how to simplify this code on its own, since the output is dependent on the effect. However, we can exactly bound the algorithm $\lambda l.\ (\mathit{sublist}\ l; \mathsf{ret}(\star))$ that ignores the returned list by the binomial distribution, given in Fig. 9. Concretely, we can show that $\lambda l.\ (\mathit{sublist}\ l; \mathsf{ret}(\star)) = \lambda l.\ \mathit{binomial}\ |l|$. As a lemma, we have that $\mathit{binomial} \leq \lambda n.\ \mathsf{step}^n(\mathsf{ret}(\star))$. We may use this to upper bound *sublist*, *i.e.* $\lambda l.\ (\mathit{sublist}\ l; \mathsf{ret}(\star)) \leq \lambda l.\ \mathsf{step}^{|l|}(\mathsf{ret}(\star))$. ⌟

3.2.3 Global State. Finally, we consider the global state effect, specified by a signature in Fig. 10. For simplicity, we only provide a single global mutable cell.

$$\begin{aligned}
&\mathsf{get} : \{X : \mathsf{tp}^{\ominus}\}\ (\mathsf{tm}^{+}(S) \to \mathsf{tm}^{\ominus}(X)) \to \mathsf{tm}^{\ominus}(X)\\
&\mathsf{set} : \{X : \mathsf{tp}^{\ominus}\}\ \mathsf{tm}^{+}(S) \to \mathsf{tm}^{\ominus}(X) \to \mathsf{tm}^{\ominus}(X)\\
&\mathsf{get}/\mathsf{get} : \{X, e\}\ \mathsf{get}(s_1.\, \mathsf{get}(s_2.\, e\ s_1\ s_2)) = \mathsf{get}(s.\, e\ s\ s)\\
&\mathsf{get}/\mathsf{set} : \{X, e\}\ \mathsf{get}(s.\, \mathsf{set}(s;\, e)) = e\\
&\mathsf{set}/\mathsf{get} : \{X, e\}\ \mathsf{set}(s;\, \mathsf{get}(s'.\, e\ s')) = \mathsf{set}(s;\, e\ s)\\
&\mathsf{set}/\mathsf{set} : \{X, e\}\ \mathsf{set}(s_1;\, \mathsf{set}(s_2;\, e)) = \mathsf{set}(s_2;\, e)\\
&\mathsf{get}/\mathsf{step} : \{X, c, e\}\ \mathsf{step}^{c}_{X}(\mathsf{get}(s.\, e\ s)) = \mathsf{get}(s.\, \mathsf{step}^{c}_{X}(e\ s))\\
&\mathsf{set}/\mathsf{step} : \{X, c, s, e\}\ \mathsf{step}^{c}_{X}(\mathsf{set}(s;\, e)) = \mathsf{set}(s;\, \mathsf{step}^{c}_{X}(e))
\end{aligned}$$

Fig. 10. Specification of the get and set primitives for single-cell global state [Plotkin and Power 2002] with a fixed state type $S : \mathsf{tp}^{+}$. The laws for interaction with the step primitive are included. Like in Fig. 4, the laws for interactions with computation type primitives are omitted for brevity, as they are analogous to those described in Section 2.7.

$$\begin{aligned}
&\mathit{twice} : \mathsf{U}(\mathsf{F}(\mathsf{nat})) \to \mathsf{F}(\mathsf{nat})\\
&\mathit{twice}\ e =\\
&\quad \mathsf{bind}\, x_1 \leftarrow e\ \mathsf{in}\\
&\quad \mathsf{bind}\, x_2 \leftarrow e\ \mathsf{in}\\
&\quad \mathsf{ret}(x_1 + x_2)
\end{aligned}$$

Fig. 11. Implementation of the *twice* function which takes a suspended computation as input, runs it twice, and adds the results. No cost is instrumented explicitly, but e may incur cost (and/or other effects).

Example 3.10 (State-dependent cost). For this example, let state type $S = \mathsf{nat}$. Recall the *double* function from Example 3.1, and consider the program

$$e \coloneqq \mathsf{get}(n.\, \mathsf{bind}\, n' \leftarrow \mathit{double}\ n\ \mathsf{in}\ \mathsf{set}(n';\, \mathsf{ret}(n)))$$

that doubles the global state and returns its original value. Here, pervasive effects are used, so they must appear in the bounding program; notice that the cost even depends on the result of the get operation. A tight bound for e is $e_{\mathrm{bound}} \coloneqq \mathsf{get}(n.\, \mathsf{set}(2n;\, \mathsf{step}^{n}(\mathsf{ret}(n))))$, specifying that e must read the global state n, set the state back to $2n$, incur n cost, and then return n; thus $e = e_{\mathrm{bound}}$. ⌟

This example illustrates once again that for general effectful programs, the effects must be available in the language of cost bounds, thus shattering the illusion that there can be a simple "shadow language" of cost bounds as discussed in Section 1.5.

3.3 Higher-Order Functions

Thus far we have considered *first-order* functions, where the function inputs were simple data. What about *higher-order* functions that takes in suspended computations as input? Given the definitions of hasCost and isBounded from Section 1, it is unclear what a bound for a higher-order function should be, especially if the input computation is effectful. Using program equality and inequality, though, a bound for a higher-order function is just another higher-order function.

Example 3.11 (Twice-run computation). In Fig. 11, we define a function *twice* that takes as input a suspended computation $e : \mathsf{U}(\mathsf{F}(\mathsf{nat}))$, runs it twice, and sums the results. In general, e could be costly, probabilistically sample, interact with mutable state, and more. Therefore, the only plausible choice of bound for e would be e itself, since $e \le e$. This aligns with standard practice in algorithms literature: for arbitrary computation inputs, the cost bound and behavioral correctness of a higher-order function depend on the specific implementation details of the program.

$$
\begin{array}{l}
\mathit{map} : \mathsf{U}(\mathsf{nat} \to \mathsf{F}(\mathsf{nat})) \to \mathsf{list}(\mathsf{nat}) \to \mathsf{F}(\mathsf{list}(\mathsf{nat})) \\
\mathit{map}\ f\ [] = \mathsf{ret}([]) \\
\mathit{map}\ f\ (x :: xs) = \\
\quad \mathsf{bind}\ ys \leftarrow \mathit{map}\ f\ xs\ \mathsf{in} \\
\quad \mathsf{bind}\ y \leftarrow f\ x\ \mathsf{in} \\
\quad \mathsf{ret}(y :: ys)
\end{array}
$$

Fig. 12. Implementation of the *map* function on lists, which applies a suspended function elementwise to an input list. No cost is instrumented explicitly, but the applications of f may incur cost (and/or other effects).

If some properties about input e are known, though, we may be able to simplify. For example, if $e; \mathsf{ret}(\star) \leq \mathsf{step}^1(\mathsf{ret}(\star))$, we can prove that $\mathit{twice}\ e; \mathsf{ret}(\star) \leq \mathsf{step}^2(\mathsf{ret}(\star))$. In other words, if we know that e incurs at most 1 cost and always returns, we can show that $\mathit{twice}\ e$ incurs at most 2 cost and always returns. ⌟

Example 3.12 (List map). In a similar direction, consider the implementation of the map function on lists in Fig. 12. If nothing is known about the input f, then *map* is the only reasonable bound for itself. However, if some properties about f are known, we can provide a more concise bound.

(1) Suppose for all x, it is the case that $f\ x; \mathsf{ret}(\star) \leq \mathsf{step}^c(\mathsf{ret}(\star))$, for some fixed cost c. Then for all lists l, we have $\mathit{map}\ f\ l; \mathsf{ret}(\star) \leq \mathsf{step}^{c|l|}(\mathsf{ret}(\star))$. In other words, if each application of f incurs at most c cost and returns, we can show that $\mathit{map}\ f\ l$ incurs at most $c|l|$ cost and always returns, the standard bound on *map* for total functions.
(2) Suppose for all x, it is the case that $f\ x; \mathsf{ret}(\star) \leq \mathit{binomial}\, n$, for some fixed n. Then, for all lists l, we have $\mathit{map}\ f\ l; \mathsf{ret}(\star) \leq \mathit{binomial}\ (n|l|)$. In other words, if the cost of each application of f is bounded by the binomial distribution with n trials, we can show that $\mathit{map}\ f\ l$ is bounded by the binomial distribution with $n|l|$ trials. ⌟

This style of reasoning aligns well with existing on-paper techniques. If details about an input computation are known, then a concise and insightful bound can be derived. Otherwise, one must examine the program in its entirety to understand the behavior.

3.4 Parallelism

When **calf** is instantiated with the *parallel cost monoid* in the sense of Niu et al. [2022a], one obtains a theory for reasoning about a version of fork-join parallelism. Parallel composition is represented via an operation $\| : \mathsf{F}(A) \times \mathsf{F}(B) \to \mathsf{F}(A \times B)$ satisfying the law $(\mathsf{step}^{c_1}(\mathsf{ret}(a)) \,\|\, \mathsf{step}^{c_2}(\mathsf{ret}(b))) = \mathsf{step}^{c_1 \otimes c_2}(\mathsf{ret}(a, b))$, where we write $c_1 \otimes c_2$ for parallel cost composition. In the **decalf** theory equipped with only the cost effect, one may continue to define and reason about parallel programs in this fashion. Moreover, because of the presence of the preorder structure, we automatically obtain a cost refinement lemma for parallel composition by monotonicity, in the sense that $e_1 \,\|\, e_2 \leq e_1' \,\|\, e_2'$ whenever $e_1 \leq e_1'$ and $e_2 \leq e_2'$. The interaction of parallelism with other effects is a difficult problem; we leave a proper theory to future work.

4 A PRESHEAF MODEL OF decalf

Our goal is to construct a model of type theory that contains a non-trivial interpretation of the constructs of **decalf**: this must necessarily contain a universe of types equipped with a built-in preorder structure, as well as a phase distinction — such that in the "extensional" phase, the inequality relations collapse to equalities. Although the technical development of this model brings together many sophisticated tools from category theory, the main ideas behind our construction

can and should be explained intuitively. Proofs of these and more results appear in the appendix of the extended version [Grodin et al. 2023].

Main Idea 1 (An interval type for automatic monotonicity). The first problem to solve when building a model of **decalf** is to devise a binary relation $(\sqsubseteq_A) \subseteq A \times A$ on every type A such that any function $A \to B$ is *automatically* monotone. The solution to this problem, which was first discovered in the world of synthetic domain theory, is to define $(\sqsubseteq_A)$ uniformly in A by considering functions into A from an ***interval***, *i.e.* some special type $\mathbb{I}$ equipped with two constants $0, 1 : \mathbb{I}$.

An interval $(\mathbb{I}, 0, 1)$ always induces a ***path relation*** $x \sqsubseteq_A y \iff \exists p : \mathbb{I} \to A. p0 = x \wedge p1 = y$, and this relation is automatically preserved by any function $f : A \to B$. Suppose that $x \sqsubseteq_A y$ and we wish to show that $fx \sqsubseteq_B fy$; by definition, we may choose some $p : \mathbb{I} \to A$ such that $p0 = x$ and $p1 = y$; then the map $f \circ p : \mathbb{I} \to B$ satisfies $p0 = fx$ and $p1 = fy$, and so we have $fx \sqsubseteq_B fy$.

Although the idea of an interval lets us define a binary relation on every type A, this relation does not enjoy almost any of the properties that we need in order to model **decalf**:

(1) **Extensional discreteness:** $\bigcirc(x \sqsubseteq_A y)$ does not necessarily imply $\bigcirc(x = y)$.
(2) **Path transitivity:** It need not be the case that $(\sqsubseteq_A)$ is transitive, *i.e.* exhibit A as a preorder.
(3) **Pointwise order:** It need not be the case that functions have the pointwise order, *i.e.* we do not necessarily have $f \sqsubseteq_{A \to B} g$ if and only if $\forall x : A. fx \sqsubseteq_B gx$.

We can solve all the problems above by using the categorical notion of *orthogonality*.

Main Idea 2 (Orthogonality). Let A be a type, and let $i : U \to V$ be a function; the concept of *orthogonality* is one way to make precise the idea that A behaves "as if" the map $i : U \to V$ were an isomorphism. We say that A is orthogonal to $i : U \to V$ when any function $f : U \to A$ can be extended to a *unique* function $i_! f : V \to A$ such that $i_! f \circ i = f$. More succinctly, the precomposition map $(- \circ i) : (V \to A) \to (U \to A)$ is required to be an isomorphism.

Many common structures may be characterized via orthogonality conditions. For instance, a set is subsingleton if and only if it is orthogonal to the map $* : \mathbf{2} \to \mathbf{1}$. Likewise, a poset is a complete partial order (cpo) if and only if it is orthogonal to $\omega \hookrightarrow \overline{\omega}$, where ω is the poset of natural numbers with the usual order and $\overline{\omega}$ is the free extension of ω by an infinite point.

We can now summarize how orthogonality solves the three problems we identified above.

Main Idea 3 (Extensional discreteness by orthogonality). We can force $\bigcirc(x \sqsubseteq_A y)$ to imply $\bigcirc(x = y)$ by refining our specification of the interval: in particular, we shall require the interval $\mathbb{I}$ to be orthogonal to the unique function $\bot \to \P_{\text{ext}}$. By our metaphor (Main Idea 2), this means that we want the interval $\mathbb{I}$ to "think" that $\P_{\text{ext}}$ is false. From a mathematical point of view, this is equivalent to saying that the $\bigcirc\mathbb{I} \cong \mathbf{1}$ as we shall have $\bigcirc\mathbb{I} = (\P_{\text{ext}} \to \mathbb{I}) \cong (\bot \to \mathbb{I}) \cong \mathbf{1}$. To see that this condition suffices, we observe that $\bigcirc(x \sqsubseteq_A y)$ is defined to be $\P_{\text{ext}} \to \exists p : \mathbb{I} \to A. p0 = x \wedge p1 = y$; this is equivalent to $\P_{\text{ext}} \to \exists p : (\P_{\text{ext}} \to \mathbb{I}) \to A. p(\lambda\_.0) = x \wedge p(\lambda\_.1) = y$. As $(\P_{\text{ext}} \to \mathbb{I}) = \bigcirc\mathbb{I} \cong \mathbf{1}$ we know that any such p must be constant, and so we have indeed have $\bigcirc(x = y)$.

There are no reasonable conditions that we can impose on the interval $\mathbb{I}$ to ensure that each path relation $(\sqsubseteq_A)$ is transitive. Indeed, Fiore and Rosolini [1997, Proposition 1.2] have shown that only a slight strengthening of the transitivity condition will imply that $\mathbb{I} \cong \mathbf{2}$, and under these conditions it would follow that every type is discrete, *i.e.* we would have $x \sqsubseteq_A y$ if and only if $x = y$. We likewise cannot hope for a condition on $\mathbb{I}$ that makes every function space have the pointwise order. In either case the best we can do is *restrict* our attention to a class of types that do have these desirable properties, and provide a universal way to approximate any given type by a type in this class; such a class of types is called a *reflective subuniverse* [Rijke et al. 2020].

Main Idea 4 (Reflective subuniverses). A reflective subuniverse is defined to be a class of types $\mathcal{S}$ such that for any type A, there exists a type $A_{\mathcal{S}} \in \mathcal{S}$ and a map $\eta_A : A \to A_{\mathcal{S}}$ to which every type $B \in \mathcal{S}$ is orthogonal.[10] Recalling our metaphor, this means that every type in $\mathcal{S}$ thinks that A is isomorphic to $A_{\mathcal{S}}$; types that do not lie in $\mathcal{S}$ would then see $A_{\mathcal{S}}$ as the "best approximation" of A by a type lying in $\mathcal{S}$. Reflective subuniverses are exponential ideals, which means that if B lies in $\mathcal{S}$, then we automatically have $(A \to B) \in \mathcal{S}$; in fact, the same applies for dependent function spaces.

Main Idea 5 (Transitivity and pointwise functions by orthogonality). It happens that given some finite collection of maps $\mathcal{M}$, the class of types orthogonal to every map in $\mathcal{M}$ is a reflective subuniverse, assuming sufficiently powerful quotient and inductive types. Therefore, in order to obtain a reflective subuniverse of types A such that $(\sqsubseteq_A)$ is transitive and $(\sqsubseteq_{B\to A})$ is pointwise, it would suffice to find *orthogonality conditions* that imply these properties. Using the interval $\mathbb{I}$ and pushouts we can indeed find a pair of maps such that if A is orthogonal to both, then the relation $(\sqsubseteq_A)$ is transitive and any function space $B \to A$ has the pointwise order. Thus we obtain a reflective subuniverse whose types satisfy all the desirable properties needed by **decalf**.

Taking stock, what exactly do we need to do in order to construct a model of **decalf** along the lines of Main Ideas 1 to 5? It will be sufficient to find a model of type theory with a proposition $¶_{\text{ext}}$ and an interval object $(\mathbb{I}, 0, 1)$ such that $\mathbb{I}$ is orthogonal to $\bot \to ¶_{\text{ext}}$. From just this data, all the rest follows by the construction of reflective subuniverses from orthogonal classes. In the sections that follow we will develop the ideas explained intuitively above with more mathematical precision, culminating in an explicit construction of a *specific* model of **decalf** supporting a cartesian closed and order-preserving embedding from the category of preorders and monotone maps.

4.1 Topos-Theoretic Preliminaries

Much of this section will revolve around synthetic constructions in the internal language of an *elementary topos*, *i.e.* a cartesian closed category with finite limits and a subobject classifier. An elementary topos $\mathscr{E}$ has an *internal language*, which is a form of extensional dependent type theory with a univalent universe of propositions (subsingleton types). We will favor working type theoretically on the inside of $\mathscr{E}$ rather than diagrammatically on the outside of $\mathscr{E}$; unless we say otherwise, all statements are to be understood as internal. We refer the reader to Awodey et al. [2021]; Maietti [2005] for further discussion of this type theoretic language.

Definition 4.1. We define an *elementary QWI-topos* to be an elementary topos closed under QWI-types à la Fiore et al. [2021].

QWI-topoi are closed under a form of *quotient inductive types* dubbed QWI-types by Fiore et al. [2021], whence the name. Quotient inductive types allow the simultaneous definition of a type by generators and relations.

Example 4.2. Every category of **Set**-valued presheaves is a QWI-topos. ⌟

Orthogonality and Local Types. In the internal language of an elementary topos $\mathscr{E}$, we will state many important conditions in terms of *orthogonality* (Main Idea 2); by our convention, we shall always mean *internal orthogonality*. We shall introduce an intermediate notion of *suborthogonality*.

Definition 4.3. Let A be a type, and let $i : U \to V$ be an arbitrary map. We say that A is *suborthogonal* (resp. *orthogonal*) to $i : U \to V$ when the induced precomposition map $A^i : A^V \to A^U$ is a monomorphism (resp. isomorphism). If J is a type parameterizing a family of maps

[10] This is in fact a universal property, as it can be seen that the pair $(A_{\mathcal{S}}, \eta_A : A \to A_{\mathcal{S}})$ are unique up to unique isomorphism with this property.

$\mathcal{S} = \{i_j : U_j \to V_j\}_{j \in J}$, we say that A is orthogonal to $\mathcal{S}$ if and only if it is orthogonal to each $i_j : U_j \to V_j \in \mathcal{S}$ for $j : J$. We will also say that A is *$\mathcal{S}$-local* when it is orthogonal to $\mathcal{S}$.

When $\mathscr{E}$ is an elementary QWI-topos, the $\mathcal{S}$-local types are *internally reflective* in $\mathscr{E}$, meaning that every type has a "best approximation" by an $\mathcal{S}$-local type.

Proposition 4.4 (Orthogonal reflection). *Let $\mathscr{E}$ be an elementary QWI-topos. In the internal language of $\mathscr{E}$, let $\mathcal{S}$ be a J-indexed family of maps for some type J; then for any type A, we may define an $\mathcal{S}$-local type $A_{\mathcal{S}}$ and a map $\eta_A : A \to A_{\mathcal{S}}$ such that every $\mathcal{S}$-local type is orthogonal to η_A.*

Proof. The orthogonal reflection can be constructed by means of a quotient inductive type, adapting the method of Rijke et al. [2020] from homotopy type theory to extensional type theory. □

4.2 Synthetic Preorders in the Presence of an Interval

Let $\mathscr{E}$ be an elementary topos equipped with an *interval object*, *i.e.* an object $\mathbb{I}$ with two elements $0, 1 : \mathbb{I}$.[11] We will work in the internal language of $\mathscr{E}$ for the remainder of this section.

Definition 4.5 (Partial order on the interval). We have an embedding $[-] : \mathbb{I} \hookrightarrow \Omega$ sending $i : \mathbb{I}$ to the proposition $(i = 1)$, which creates a partial order $i \to_{\mathbb{I}} j \iff ([i] \to [j])$ on $\mathbb{I}$.

Definition 4.6 (Finite chains). For each finite ordinal n, we may define an auxiliary figure $\mathbb{I}_n$ classifying "chains" of length n in $\mathbb{I}$, setting $\mathbb{I}_n$ to be the subobject of $\mathbb{I}^n$ spanned by vectors $(i_0 \to_{\mathbb{I}} \ldots \to_{\mathbb{I}} i_{n-1})$. In particular, we have $\mathbb{I}_0 = \mathbf{1}$, $\mathbb{I}_1 = \mathbb{I}$, and $\mathbb{I}_2 = \{i, j : \mathbb{I} \mid i \to_{\mathbb{I}} j\}$.

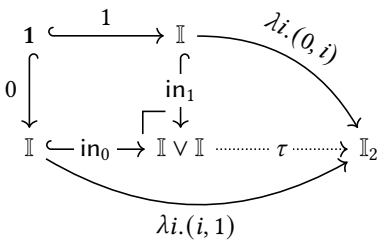
4.2.1 Paths and Path-Transitivity.

Definition 4.7 (Paths). A *path* in a type A from an element $x : A$ to $y : A$ is defined to be a function $\alpha : \mathbb{I} \to A$ such that $\alpha 0 = x$ and $\alpha 1 = y$.

The following definitions are adapted from Fiore and Rosolini [1997].

Definition 4.8 (The path relation). On any type A we may define a reflexive relation $\sqsubseteq_A \subseteq A \times A$, saying that $x \sqsubseteq_A y$ if and only if there *exists* some path from x to y in A.

Definition 4.9 (Path-transitivity). A type A is called *path-transitive* when it is orthogonal to the dotted cocartesian gap map $\tau : \mathbb{I} \vee \mathbb{I} \to \mathbb{I}_2$ depicted below:



Lemma 4.10 (Path-transitive types are preorders). *If a type A is path-transitive in the sense of Definition 4.9, then the path relation $\sqsubseteq_A$ is a preorder on A.*

As remarked by Fiore and Rosolini [2001, §1.3], the converse to Lemma 4.10 need not hold.

[11] We do not here assume that $0 \neq 1$.

4.2.2 Boundary Separation. As it stands, there could be two distinct paths $\mathbb{I} \to A$ that have the same endpoints. We wish to isolate the types A for which paths are uniquely determined by their endpoints; this property was dubbed *boundary separation* by Sterling et al. [2019, 2022] and Σ-separation by Fiore and Rosolini [2001]. The purpose of imposing boundary separation in our setting is to make the path order on function spaces *pointwise*, as we shall see in Lemma 4.15.

Definition 4.11. A type A is called *boundary separated* when it is *suborthogonal* to the boundary inclusion $i : \mathbf{2} \to \mathbb{I}$ determined by the two endpoints of $\mathbb{I}$.

Although the presentation of boundary separation in terms of suborthogonality is simple and elegant, it will be later advantageous to observe that boundary separation can also be seen as an orthogonality property so as to incorporate it into a *reflective* subcategory via Proposition 4.4. To that end, we introduce *path suspensions* below in order to state Lemma 4.13, which characterizes boundary separation in terms of orthogonality.

Definition 4.12. We define the *path suspension* of a type A to be following pushout:

$$\begin{array}{ccc} A \times \mathbf{2} & \xrightarrow{A \times [0 \mid 1]} & A \times \mathbb{I} \\ {\scriptstyle \pi_2}\downarrow & & \downarrow{\scriptstyle \mathsf{m}} \\ \mathbf{2} & \xrightarrow[[\mathsf{n} \mid \mathsf{s}]]{} & \mathbb{S}A \end{array}$$

The universal property of path suspension places functions $\mathbb{S}A \to B$ in bijection with triples $(n, s : B; \beta : A \to B^{\mathbb{I}})$ where β is valued in paths from n to s.

Lemma 4.13. *A type A is boundary separated if and only if it is orthogonal to the path suspension $\mathbb{S}* : \mathbb{S}\mathbf{2} \to \mathbb{S}\mathbf{1}$ of the terminal map $* : \mathbf{2} \to \mathbf{1}$.*

Remark 4.14. Observe that a type is a *proposition* (subsingleton) if and only if it is orthogonal to the terminal map $* : \mathbf{2} \to \mathbf{1}$. Boundary separation asserts that the path spaces with fixed endpoints are propositions. Our use of *path suspension* therefore corresponds precisely to the observation of Christensen et al. [2020] in the context of homotopy type theory that a type is *separated* with respect to a given class of maps if and only if it is *local* (orthogonal) to their suspensions. Fiore and Rosolini [2001] equivalently describe boundary separation in terms of orthogonality to a different (but closely related) map; we have preferred here to make the connection with suspensions.

Lemma 4.15. *Let $x : A \vdash Bx$ be a family of boundary separated types, and let $f, g : (x : A) \to Bx$ be a pair of dependent functions. Then $f \sqsubseteq_{(x:A)\to Bx} g$ if and only if for all $x : A$ we have $fx \sqsubseteq_{Bx} gx$.*

4.2.3 Synthetic Preorders. We now come to a suitable definition of "synthetic preorder" within $\mathscr{E}$.

Definition 4.16. A type A is called a *synthetic preorder* when it is both path-transitive and boundary separated, *i.e.* orthogonal to both $\tau : \mathbb{I} \vee \mathbb{I} \to \mathbb{I}_2$ and $\mathbb{S}* : \mathbb{S}\mathbf{2} \to \mathbb{S}\mathbf{1}$.

The benefit of defining synthetic preorders in terms of (internal) orthogonality is that they automatically form a (full, internal) reflective subcategory of $\mathscr{E}$ via Proposition 4.4; we specialize the general result below to the case of path-transitivity.

Corollary 4.17 (Synthetic preorder reflection). *Assume that $\mathscr{E}$ is a QWI-topos. For any type A we may define a synthetic preorder* PA *equipped with a map $\eta_A : A \to$ PA to which any synthetic preorder B is orthogonal. In other words, every type A has a* reflection PA *as a synthetic preorder.*

Corollary 4.17 further implies that the full (internal) subcategory of $\mathscr{E}$ spanned by synthetic preorders is cartesian closed, and moreover closed under dependent functions spaces for families of synthetic preorders. It is also (internally) complete and cocomplete, with limits computed as in the ambient category and colimits computed by applying the synthetic preorder reflection to the those of the ambient category. These results concerning (internally) orthogonal subcategories, among many others, can be found in the work of Rijke [2019]; Rijke et al. [2020].

4.2.4 Discrete Types. We introduce a notion of *discrete* types that provides a sufficient condition for being a synthetic preorder.

Definition 4.18. A type A is called *discrete* when the following equivalent conditions hold:

(1) the type A is orthogonal to the map $\mathbb{I} \to \mathbf{1}$;
(2) the type A is boundary separated and the path relation $(\sqsubseteq_A)$ is the diagonal, *i.e.* $(\sqsubseteq_A) \subseteq (=)$.

Proposition 4.19. *Any discrete type A is a synthetic preorder.*

4.3 A Synthetic Theory of *Partially Discrete* Preorders

Let $\mathscr{E}$ be an elementary topos equipped with an *interval object* as in the previous section. In this section, we extend the axiomatics of $\mathscr{E}$ to account for a *phase distinction* under which every synthetic preorder becomes discrete. Semantically, this corresponds to assuming an indeterminate proposition $\P_{\mathrm{ext}} : \Omega$ such that the interval $\mathbb{I}$ is $\P_{\mathrm{ext}}$*-connected* in the sense of Rijke et al. [2020], *i.e.* such that the proposition $\P_{\mathrm{ext}} \to (\mathbb{I} \cong \mathbf{1})$ holds or, equivalently, $\mathbb{I}$ is orthogonal to $\bot \to \P_{\mathrm{ext}}$. The purpose of this axiom is as follows: if $\mathbb{I}$ is $\P_{\mathrm{ext}}$-connected in the above sense, any path $\mathbb{I} \to A$ restricts to a *constant* function under $\P_{\mathrm{ext}}$; thus we always have $\P_{\mathrm{ext}} \wedge x \sqsubseteq_A y \to x = y$. Thus in the presence of this axiom, any synthetic preorder is "partially discrete" in the sense of being discrete as soon as $\P_{\mathrm{ext}} = \top$.

We now summarize the axiomatics of *synthetic partially discrete preorder theory* for an arbitrary elementary QWI-topos $\mathscr{E}$.

Axiom 4.20. *We assume a proposition $\P_{\mathrm{ext}} : \Omega$.*

Axiom 4.21. *We assume a type $\mathbb{I}$ together with two elements $0, 1 : \mathbb{I}$.*

Axiom 4.22. *The interval object $\mathbb{I}$ of Axiom 4.21 is $\P_{\mathrm{ext}}$-connected.*

Axiom 4.23. *The natural numbers object $\mathbb{N}$ is discrete in the sense of Definition 4.18.*

Definition 4.24. Let $\mathscr{E}$ be an elementary QWI-topos.

(1) We say that $\mathscr{E}$ is a *model of synthetic preorder theory* when it satisfies Axioms 4.21 and 4.23.
(2) If $\mathscr{E}$ additionally satisfies Axioms 4.20 and 4.22, then we call it a *model of synthetic partially discrete preorder theory*.

The purpose of making $\mathbb{N}$ a discrete type is to allow the motive of the elimination principle of inductive data types to range over arbitrary types, which enables us to compute program invariants in **decalf** using the dependent sum of the LF as mentioned in Section 1.5.

4.4 Well-Adapted Models

A priori, a cost bound in the theory of synthetic preorders has little to do with a cost bound in concrete preorders. However, in a model of synthetic preorders that embeds the concrete preorders, we may relate a synthetic cost bound to a more traditional account of cost bounds in concrete preorders. This is captured by the notion of *well-adapted models*.

Definition 4.25. Let $\mathscr{E}$ be a model of synthetic preorder theory.

(1) We say that $\mathscr{E}$ is *well-adapted* when there is a fully faithful cartesian closed functor **Preord** $\hookrightarrow \mathscr{E}$ that preserves the interval object. This means that the actual interval $[1] : \mathbf{Preord}$ is sent to $\mathbb{I}$ and the points $0, 1 : \mathbb{I}$ are determined by the maps $0, 1 \colon [0] \to [1]$ respectively.
(2) If moreover $\mathscr{E}$ is a model of synthetic *partially discrete* preorder theory, we say $\mathscr{E}$ is well-adapated if there is a functor satisfying the conditions above and whose image lies within the $\P_{\text{ext}}$-connected types, *i.e.* those that restrict under $\P_{\text{ext}}$ to a singleton.

This terminology is inspired by an analogous situation in *synthetic differential geometry* where a topos model $\mathscr{E}$ is called well-adapted [Dubuc 1979] when there is an embedding of ordinary manifolds into $\mathscr{E}$ preserving the differential geometry structure. The canonical example of a well-adapted model of synthetic (partially discrete) preorder theory is (augmented) simplicial sets, for which the corresponding embedding is given by the nerve functor. Now, we discuss how to relate synthetic cost bounds to cost bounds in concrete preorders in well-adapted models.

Well-Adapted Models and Concrete Preorders. Let $\mathscr{E}$ be a well-adapted model of synthetic preorder theory (Definition 4.25), and write $N \colon \mathbf{Preord} \hookrightarrow \mathscr{E}$ for the associated embedding.

Theorem 4.26. *Let $(P, \leq)$ be a concrete preorder. If $p \leq q$, then $Np \sqsubseteq_{NP} Nq$ holds in $\mathscr{E}$.*

Corollary 4.27 (Completeness of synthetic cost bounds). *Let $P, Q : \mathbf{Preord}$ be concrete preorders. Given monotone maps $f, g \colon P \to Q$ such that $f \leq g$ on the pointwise order, then the synthetic preorder relation $Nf \sqsubseteq_{NP \to NQ} Ng$ holds in $\mathscr{E}$.*

Proof. By Theorem 4.26 and the fact that e is cartesian closed. □

Theorem 4.28. *Let $(P, \leq)$ be a concrete preorder. If $x \sqsubseteq_{NP} y$ holds in $\mathscr{E}$, then there exist $p, q : P$ such that $p \leq q$ and $x = Np$ and $y = Nq$.*

Corollary 4.29 (Soundness of synthetic cost bounds). *Let $P, Q : \mathbf{Preord}$ be concrete preorders. Given maps $x, y \colon NP \to NQ$ such that the synthetic preorder relation $x \sqsubseteq_{NP \to NQ} y$ holds in $\mathscr{E}$, then there exist $f, g \colon P \to Q$ such that $f \leq g$ in the pointwise order and $x = Nf$ and $y = Ng$.*

Proof. By Theorem 4.28 and the fact that N is cartesian closed. □

4.5 Algebra Models of decalf in Synthetic Partially Discrete Preorder Theory

In this section, we describe how to instantiate the constructs of **decalf** in any model $\mathscr{E}$ of synthetic partially discrete preorder theory in the sense of Definition 4.24. To construct our model, we fix universes $\mathcal{U} \in \mathcal{V}$ in $\mathscr{E}$; judgments of **calf** are intepreted in the outer universe $\mathcal{V}$. We will write $\mathcal{U}_{\text{P}} \subseteq \mathcal{U}$ for the subuniverse of $\mathcal{U}$ spanned by synthetic preorders, *i.e.* path-transitive and boundary separated types.

4.5.1 Cost Structure. The theory of **decalf** is parameterized by a cost monoid $\mathbb{C} : \mathcal{U}_{\text{P}}$ that is $\P_{\text{ext}}$-connected, *i.e.* becomes a singleton when $\P_{\text{ext}}$ is true. In a well-adapted model of synthetic preorder theory $\mathscr{E}$ in the sense of Section 4.3, we may take an ordinary preordered monoid P and define $\mathbb{C}$ as the image of P under the full embedding **Preord** $\hookrightarrow \mathscr{E}$. An alternative method would be to define the cost monoid $\mathbb{C}$ as a quotient inductive type that builds in the expected order structure. For instance, we may define the synthetic preorder ω that can be thought of as the colimit of the inclusions of finite chains $\mathbb{I}_0 \to \mathbb{I}_1 \to \dots$ by means of a quotient inductive type, shown in Fig. 13. The object ω as defined is $\P_{\text{ext}}$-connected because $\mathbb{I}$ is.

4.5.2 Monads for Effects. Let M be a (strong) monad on $\mathcal{U}$. The monad M may not preserve the property of being a synthetic preorder, but we may adapt it to a monad M_{P} on $\mathcal{U}_{\text{P}}$ by postprocessing

data $\omega : \mathcal{U}_{\mathsf{P}}$ **where**
  zero : ω
  suc : $\omega \to \omega$
  rel : $\omega \to \mathbb{I} \to \omega$
  _ : $(n : \omega) \to \mathsf{rel}\, n\, 0 = n$
  _ : $(n : \omega) \to \mathsf{rel}\, n\, 1 = \mathsf{suc}\, n$

Fig. 13. Quotient inductive type defining cost structure ω.

with the synthetic preorder reflection, sending $A : \mathcal{U}_{\mathsf{P}}$ to $\mathsf{P}(MA)$.[12] We can then adapt M_{P} to support a cost effect using the *cost monad transformer* corresponding to the writer monad $\mathbb{C} \times -$. In particular, we define a new monad T on $\mathcal{U}_{\mathsf{P}}$ by the assignment $TA \coloneqq M_{\mathsf{P}}(\mathbb{C} \times -)$.

(1) For verifying pure code (as in **calf**), we can let M be the identity monad.
(2) For nondeterminism (as in Section 3.2.1), we can let M be the *free semilattice* monad, which can be defined by a quotient inductive type.
(3) For probabilistic choice (as in Section 3.2.2), we can let M be the *free convex space* monad, which is likewise (constructively) definable by a quotient inductive type.[13]
(4) For global state (as in Section 3.2.3), we can let M be the state monad $S \to S \times -$.

As an alternative to defining T using the cost monad transformer as above, we can also treat the effect signatures of Section 3.2 as specifications of an algebraic effect, where for each $c : \mathbb{C}$ we have a generating operation step^c. In such cases, we can simply define T as the *free* monad for these effect signature using a quotient inductive type.

Remark 4.30. Although we have restricted our attention to *Eilenberg–Moore models* of **decalf** above, it is indeed possible and desirable to consider more general models. For example, our interpretation of global state in terms of algebras for the state monad is somewhat bizarre and uncanonical; this could be replaced by interpreting computation types as $(\mathbb{C} \times -)$-algebras, and then letting $\mathsf{U}(X) \coloneqq S \to X$ and $\mathsf{F}(A) \coloneqq A \times \mathbb{C} \times S$.

4.5.3 Universes of Positive and Negative Types. We then define tp^+ to be $\mathcal{U}_{\mathsf{P}}$ itself, letting the decoding function $\mathsf{tm}^+(A) \colon \mathsf{tp}^+ \to \mathbf{Jdg}$ be the image of $A : \mathcal{U}_{\mathsf{P}}$ under the inclusion $\mathcal{U}_{\mathsf{P}} \hookrightarrow \mathcal{V}$, which we shall leave implicit in our informal notations. We interpret $\mathsf{tp}^\ominus$ by the type of T-algebras where T is the application of the cost monad transformer to a monad M from Section 4.5.2, which we may equip, as we please, with the structure of the Eilenberg–Moore category. Then $\mathsf{tm}^\ominus(X)$ is interpreted the same as $\mathsf{tm}^+(\mathsf{U}(X))$. Thus we have a free-forgetful adjunction $\mathsf{F} \dashv \mathsf{U} \colon \mathsf{tp}^\ominus \to \mathsf{tp}^+$ interpreting the call-by-push-value adjunctive structure of **decalf**.

4.5.4 Inequality Relation. For any type $A : \mathsf{tp}^+$ and elements $x, y : \mathsf{tm}^+(A)$, the inequality $x \leq y$ is interpreted by the path relation $x \sqsubseteq_{\mathsf{tm}^+(A)} y$, which is transitive because A lies in $\mathcal{U}_{\mathsf{P}}$. The $\leq_{\mathsf{ext}}$ axiom is a consequence of the partial discreteness axiom (Axiom 4.22) that we have assumed of $\mathscr{E}$; the $\leq_{\mathsf{mono}}$ axiom holds by definition. The $\leq_{\mathsf{pi}}$ axiom holds by Lemma 4.15, as every synthetic preorder is boundary separated. Thus the path relation $x \sqsubseteq_{\mathsf{tm}^+(A)} y$ is a proposition (and so a synthetic preorder) and may be internalized as a type of **decalf**.

[12] If M does preserve synthetic preorders, then we have $M_{\mathsf{P}} \cong M$ as reflections are idempotent.
[13] Classically this is equivalent to the finite distribution monad; in our definitions we must be careful to give constructive definitions because the logic of $\mathscr{E}$ is not classical.

4.6 A Presheaf Model of decalf in Augmented Simplicial Sets

We will construct a presheaf model of **decalf** by equipping a simplicial model of synthetic preorders with a phase distinction.

4.6.1 Simplicial Sets for Synthetic Preorders. To model synthetic preorders in a QWI-topos, we take a cue from higher category theory and consider *simplicial sets*, which are presheaves on the simplex category defined below.

Definition 4.31 (Simplex category). We will write $\mathbf{\Delta}$ for the *simplex category*, *i.e.* the category of inhabited finite ordinals $[n]$ and order-preserving maps between them. By convention, $[0]$ will denote the singleton ordinal, *i.e.* the terminal object of $\mathbf{\Delta}$.

What do simplicial sets have to do with preorders? Every preorder can be reconstructed by gluing simplices together in a canonical way; this is the *density* of the embedding $I\colon \mathbf{\Delta} \hookrightarrow \mathbf{Preord}$, which implies that the corresponding nerve functor $N\colon \mathbf{Preord} \to \mathrm{Pr}(\mathbf{\Delta})$ sending a preorder P to the restricted hom presheaf $\hom_{\mathbf{Preord}}(I-, P)$ is fully faithful. In this way, simplicial sets are an appropriate place to study concrete preorders; this is the content of a well-adapted model as defined in Section 4.3. The synthetic preorder theory of simplicial sets, then, studies sufficient conditions in the internal language for arbitrary simplicial sets to "behave like" those that arise from actual preorders via the nerve functor N.

Theorem 4.32. *The category* $\mathrm{Pr}(\mathbf{\Delta})$ *of simplicial sets forms a non-trivial model of* synthetic preorder theory *in the sense of Definition 4.24 in which the interval is given by the representable presheaf* $\mathbf{y}[1]$ *and its two global points.*

4.6.2 Augmented Simplicial Sets for Synthetic Partially Discrete Preorders. In fact, $\mathrm{Pr}(\mathbf{\Delta})$ also forms a (highly degenerate) model of *synthetic partially discrete preorder theory* in the sense of Definition 4.24, setting $\P_{\mathrm{ext}} := \bot$. Our goal is to find a *non-trivial* model $\mathscr{E}$, *i.e.* where the slice $\mathscr{E}/\P_{\mathrm{ext}}$ is not the terminal category. The most canonical choice for such a topos is obtained by *freely extending* $\mathrm{Pr}(\mathbf{\Delta})$ with a maximal topos-theoretic point by forming an "inverted Sierpiński cone", *i.e.* the Artin gluing [Artin et al. 1972] of the constant presheaves functor $\mathbf{Set} \to \mathrm{Pr}(\mathbf{\Delta})$. This gluing can also be presented equivalently by presheaves on a different category, as adding a maximal point to a presheaf topos corresponds (dually) to freely extending the base category by an initial object — which amounts in the case of $\mathbf{\Delta}$ to the use of *augmented simplicial sets*. These two perspectives on the same topos are both useful, and play a role in our results.

Definition 4.33 (Augmented simplex category). We will write $\mathbf{\Delta}_\bot$ for the *augmented simplex category*, the free extension of $\mathbf{\Delta}$ by an initial object $[-1]$. Concretely, $\mathbf{\Delta}_\bot$ can be thought of as the category of *arbitrary* finite ordinals and order-preserving maps between them; under this interpretation, $[-1]$ corresponds to the empty ordinal.

Theorem 4.34. *The category* $\mathrm{Pr}(\mathbf{\Delta}_\bot)$ *of* augmented simplicial sets *forms a non-trivial well-adapted model of* synthetic partially discrete preorder theory *in the sense of Definition 4.24 in which:*

(1) *the interval is given by the representable presheaf* $\mathbb{I} := \mathbf{y}[1]$*;*
(2) *and the phase distinction is given by the representable subterminal presheaf* $\P_{\mathrm{ext}} := \mathbf{y}[-1]$.

*4.6.3 Soundness of **decalf**.* The following soundness theorem is a corollary of Theorem 4.34 via the description of algebra models of **decalf** in Section 4.5.

Theorem 4.35 (Soundness). *For any of the notions of computational effect considered in Section 3.2, we have a non-trivial model of the **decalf** theory in* $\mathrm{Pr}(\mathbf{\Delta}_\bot)$.

5 CONCLUSION

In this work, we presented **decalf**, an inequational extension of **calf** [Niu et al. 2022a] that supports precise and approximate bounds on the cost and effect structure of programs. Ab initio, the theory of **decalf** has been forged and guided by the pragmatic struggles encountered in cost analysis and program verification. Throughout the development, our guiding principle has been that a *cost bound for an effectful program should be another effectful program.*

In Section 3, we demonstrated this methodology through a variety of case studies. For pure, first-order algorithms, we were able to provide simple proofs of combined cost and correctness. Such proofs in **decalf** are more streamlined than their **calf** counterparts and can be carried out without reference to any separable notion of recurrence. Instead, the code itself serves the role of the recurrence, which we then solve for a closed form, either exactly using equality or loosely using inequality. Then, using the extensional modality, we were able to extract extensional equalities from both equality and inequality program bounds. For example, from the cost and correctness proofs of various sorting algorithms, we may determine immediately that all the given sorting algorithms are extensionally equal. This approach scaled naturally to support more complex classes of programs, including higher-order programs with non-cost effects.

In Section 4, we justified this style of reasoning using the notion of a *synthetic partially discrete preorder theory*, a novel formulation of an *intrinsic* theory of preorders that smoothly integrates with the existing intension-extension phase distinction of **calf**. To obtain a model of this new theory, we draw inspiration from both work in directed type theory and synthetic domain theory and characterize the synthetic preorders using simple orthogonality conditions, which furnish a well-behaved subuniverse that supports the structures for workaday program verification.

6 FUTURE WORK

We view **decalf** as a starting point for deeper investigations. Here, we outline future directions.

Amortized analysis via coinduction. Our approach draws inspiration from Grodin and Harper [2023], who perform amortized analyses in **calf** for programs whose only effect is cost. We anticipate that **decalf** could be used to perform a broader class of amortized analyses.

Advanced probabilistic reasoning. In Section 3.2.2, we implement a simple probabilistic program and show an exact bound on its cost. However, many algorithms in practice do not have well-behaved distributions, so one may instead wish to analyze expected and high probability bounds. We believe that the techniques presented here will scale to support such reasoning in future work.

Parallelism and more sophisticated effects. As mentioned in Section 3.4, while parallelism is compatible with pure algorithms in **decalf**, we leave a proper theory of the interaction between parallelism and other effects to future work. Additionally, it would be useful to consider a semantics for **decalf** that goes beyond the simple, non-enriched algebraic effects we have considered here, allowing for constructs like control or unbounded recursion.

Abstraction and specification implementations. One drawback of **decalf**, inherited from **calf**, is that one must compute the result of a computation via a "specification" implementation to give a cost bound. Unfortunately, this puts abstraction at odds with cost analysis: in order to export a bound of an abstract computation, one must also make the return value public. We observe a similarity to frameworks based on program logics, in which one sometimes verifies an effectful (there, imperative; here, costly) algorithm by first providing a functional specification (for instance, see the case study on list fold in Iris [Birkedal and Bizjak 2022]). We hope this tension can be resolved in future iterations of **decalf**.

DATA AVAILABILITY STATEMENT

Building on the work of Niu et al. [2022b], the definition of **decalf** and the examples presented have been mechanized in the Agda proof assistant [Norell 2009]. Several case studies considered by Niu et al. [2022a] have been adapted to the **decalf** setting, as well. The source code is available as an artifact [Grodin et al. 2024] and via a GitHub repository (https://github.com/jonsterling/agda-calf).

ACKNOWLEDGMENTS

We are grateful to Marcelo Fiore, Runming Li, and Parth Shastri for many insightful discussions. Additionally, we thank the anonymous reviewers for their thoughtful comments.

This work was supported in part by AFOSR (Tristan Nguyen, program manager) under grants MURI FA9550-15-1-0053, FA9550-19-1-0216, FA9550-21-0009, and FA9550-23-1-0728 and in part by the National Science Foundation under award number CCF-1901381, and by AFRL through the NDSEG fellowship. This work was co-funded by the European Union under the Marie Skłodowska-Curie Actions Postdoctoral Fellowship grant agreement 101065303 (https://cordis.europa.eu/project/id/101065303). 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, AFRL, the European Union, or the European Commission. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

Danel Ahman, Neil Ghani, and Gordon D. Plotkin. 2016. Dependent Types and Fibred Computational Effects. In *Foundations of Software Science and Computation Structures*, Bart Jacobs and Christof Löding (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 36–54.

Michael Artin, Alexander Grothendieck, and Jean-Louis Verdier. 1972. *Théorie des topos et cohomologie étale des schémas.* Lecture Notes in Mathematics, Vol. 269, 270, 305. Springer-Verlag, Berlin. Séminaire de Géométrie Algébrique du Bois-Marie 1963–1964 (SGA 4), Dirigé par M. Artin, A. Grothendieck, et J.-L. Verdier. Avec la collaboration de N. Bourbaki, P. Deligne et B. Saint-Donat.

Steve Awodey, Nicola Gambino, and Sina Hazratpour. 2021. Kripke-Joyal forcing for type theory and uniform fibrations. (2021). arXiv:2110.14576 [math.LO] Unpublished manuscript.

Lars Birkedal and Aleš Bizjak. 2022. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. https://iris-project.org/tutorial-material.html

J. Daniel Christensen, Morgan Opie, Egbert Rijke, and Luis Scoccola. 2020. Localization in Homotopy Type Theory. *Higher Structures* 4 (Feb. 2020), 1–32. Issue 1. https://higher-structures.math.cas.cz/api/files/issues/Vol4Iss1/ChrOpiRijSco

Eduardo J. Dubuc. 1979. Sur les modèles de la géométrie différentielle synthétique. *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 20, 3 (1979), 231–279. http://eudml.org/doc/91216

Marcelo P. Fiore. 1997. An Enrichment Theorem for an Axiomatisation of Categories of Domains and Continuous Functions. *Mathematical Structures in Computer Science* 7, 5 (Oct. 1997), 591–618. https://doi.org/10.1017/S0960129597002429

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

Marcelo P. Fiore and Giuseppe Rosolini. 1997. The category of cpos from a synthetic viewpoint. In *Thirteenth Annual Conference on Mathematical Foundations of Progamming Semantics, MFPS 1997, Carnegie Mellon University, Pittsburgh, PA, USA, March 23-26, 1997 (Electronic Notes in Theoretical Computer Science, Vol. 6)*, Stephen D. Brookes and Michael W. Mislove (Eds.). Elsevier, 133–150. https://doi.org/10.1016/S1571-0661(05)80165-3

Marcelo P. Fiore and Giuseppe Rosolini. 2001. Domains in H. *Theoretical Computer Science* 264, 2 (Aug. 2001), 171–193. https://doi.org/10.1016/S0304-3975(00)00221-8

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

Harrison Grodin and Robert Harper. 2023. Amortized Analysis via Coinduction. In *10th Conference on Algebra and Coalgebra in Computer Science (CALCO 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 270)*, Paolo Baldan and Valeria de Paiva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:6. https://doi.org/10.4230/LIPIcs.CALCO.2023.23

Harrison Grodin, Robert Harper, Yue Niu, and Jonathan Sterling. 2023. Decalf: A Directed, Effectful Cost-Aware Logical Framework (Extended Version). https://doi.org/10.48550/arXiv.2307.05938 arXiv:2307.05938 [cs]

Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. 2024. `agda-calf` *v2.0.0*. https://doi.org/10.1145/3580425

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

C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (July 1961), 321. https://doi.org/10.1145/366622.366644

C. A. R. Hoare. 1962. Quicksort. *Comput. J.* 5, 1 (Jan. 1962), 10–16. https://doi.org/10.1093/comjnl/5.1.10

Martin Hofmann. 1995. *Extensional concepts in intensional type theory*. Ph. D. Dissertation. University of Edinburgh, Edinburgh.

J. M. E. Hyland. 1991. First steps in synthetic domain theory. In *Category Theory*, Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 131–156.

Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. 2019. Constructing Quotient Inductive-inductive Types. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 2:1–2:24. https://doi.org/10.1145/3290315

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

Paul Blain Levy. 2003. *Call-by-Push-Value: A Functional/Imperative Synthesis*. Kluwer, Semantic Structures in Computation, 2.

Daniel R. Licata and Robert Harper. 2011. 2-Dimensional Directed Type Theory. *Electronic Notes in Theoretical Computer Science* 276 (2011), 263–289. https://doi.org/10.1016/j.entcs.2011.09.026 Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

Maria Emilia Maietti. 2005. Modular correspondence between dependent type theories and categories including pretopoi and topoi. *Mathematical Structures in Computer Science* 15, 6 (2005), 1089–1149. https://doi.org/10.1017/S0960129505004962

Per Martin-Löf. 1984. *Intuitionistic type theory*. Studies in Proof Theory, Vol. 1. Bibliopolis. iv+91 pages.

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022a. A Cost-Aware Logical Framework. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022). https://doi.org/10.1145/3498670 arXiv:2107.04663 [cs.PL]

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022b. `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.

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

Wesley Phoa. 1991. *Domain Theory in Realizability Toposes*. Ph. D. Dissertation. University of Edinburgh.

Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*. Springer-Verlag, Berlin, Heidelberg, 342–356.

Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2017. Monadic Refinements for Relational Cost Analysis. *Proc. ACM Program. Lang.* 2, POPL, Article 36 (dec 2017), 32 pages. https://doi.org/10.1145/3158124

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

Emily Riehl and Michael Shulman. 2017. A type theory for synthetic ∞-categories. *Higher Structures* 1 (2017), 147–224. Issue 1. arXiv:1705.07442 [math.CT] https://journals.mq.edu.au/index.php/higher_structures/article/view/36

Egbert Rijke. 2019. *Classifying Types*. Ph. D. Dissertation. Carnegie Mellon University. arXiv:1906.09435

Egbert Rijke, Michael Shulman, and Bas Spitters. 2020. Modalities in homotopy type theory. *Logical Methods in Computer Science* 16 (Jan. 2020). Issue 1. https://doi.org/10.23638/LMCS-16(1:2)2020 arXiv:1706.07526 [math.CT]

Jonathan Sterling. 2021. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. Ph. D. Dissertation. Carnegie Mellon University. https://doi.org/10.5281/zenodo.6990769 Version 1.1, revised May 2022.

Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2019. Cubical Syntax for Reflection-Free Extensional Equality. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 31:1–31:25. https://doi.org/10.4230/LIPIcs.FSCD.2019.31 arXiv:1904.08562 [cs.LO]

Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2022. A Cubical Language for Bishop Sets. *Logical Methods in Computer Science* 18 (March 2022). Issue 1. https://doi.org/10.46298/lmcs-18(1:43)2022 arXiv:2003.01491 [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]

Taichi Uemura. 2021. *Abstract and Concrete Type Theories*. Ph. D. Dissertation. Universiteit van Amsterdam, Amsterdam. https://www.illc.uva.nl/cms/Research/Publications/Dissertations/DS-2021-09.text.pdf

Taichi Uemura. 2023. A general framework for the semantics of type theory. *Mathematical Structures in Computer Science* (2023), 1–46. https://doi.org/10.1017/S0960129523000208

Matthijs Vákár. 2017. *In Search of Effectful Dependent Types.* Ph. D. Dissertation. University of Oxford. https://doi.org/10.48550/arXiv.1706.07997 arXiv:1706.07997 [cs.LO]

Received 2023-07-11; accepted 2023-11-07